

supermeng

The Nagios logo is centered, featuring the word "Nagios" in a stylized, metallic blue font with a green-to-blue gradient. It is flanked by two rectangular grids of small squares, each grid being 10 squares wide and 8 squares high. The background is a solid blue with a subtle gradient and some faint, larger-scale geometric patterns.

Nagios

网络监控中文版

二〇〇八年

目 录

第 2 章 关于Nagios.....	1
2.1. 什么是Nagios?	1
2.2. 系统需求.....	1
2.3. 版权.....	1
2.4. 致谢.....	2
2.5. 下载最新版本.....	2
第 3 章 Nagios 3.0 新特性.....	2
3.1. 更新日志.....	2
3.2. 变更与新特征.....	2
第 4 章 入门.....	12
4.1. 给新手的建议.....	12
4.2. 旧Nagios升级到当前版本.....	12
4.2.1. 从旧的 3. x版本升级到当前版本	13
4.2.2. 从 2. x升级到 3. x	13
4.2.3. 从RPM包安装状态升级	14
4.3. 快速安装指南.....	15
4.3.1. 介绍	15
4.3.2. 指南	15
4.3.3. 安装后该做的	15
4.4. 基于Fedora平台的快速指南.....	15
4.4.1. 介绍	15
4.4.2. 准备软件包	16
4.4.3. 操作过程	16
4.5. 基于openSUSE平台的快速指南.....	20
4.5.1. 介绍	20
4.5.2. 所需的软件包	20
4.5.3. 操作过程	20
4.6. 基于Ubuntu平台的快速指南.....	23
4.6.1. 介绍	23
4.6.2. 所需软件包	23
4.6.3. 操作过程	24
4.7. 监控Windows主机.....	27
4.7.1. 介绍	27
4.7.2. 概览	27
4.7.3. 步骤	27
4.7.4. 已经做了什么?	28
4.7.5. 首要条件	28
4.7.6. 安装Windows代理程序	28

4.7.7. 配置Nagios	29
4.7.8. 口令保护	32
4.7.9. 重启动Nagios	32
4.8. 监控Linux/Unix主机	33
4.8.1. 介绍	33
4.8.2. 概览	33
4.9. 监控路由器和交换机	33
4.9.1. 介绍	33
4.9.2. 概览	34
4.9.3. 步骤	34
4.9.4. 已经做了什么?	34
4.9.5. 必备工作	35
4.9.6. 配置Nagios	35
4.9.7. 监控服务	35
4.9.8. 监控丢包率和RTA	36
4.9.9. 监控SNMP状态信息	36
4.9.10. 监控带宽和流量	37
4.9.11. 重启动Nagios	38
4.10. 监控网络打印机	38
4.10.1. 介绍	38
4.10.2. 概览	39
4.10.3. 步骤	39
4.10.4. 已经做了什么?	39
4.10.5. 事先准备工作	39
4.10.6. 配置Nagios	40
4.10.7. 重启动Nagios	41
4.11. 监控Netware服务器	41
4.11.1. 介绍	41
4.11.2. 概览	42
4.11.3. 其他资源	42
4.12. 监控公众服务平台	42
4.12.1. 介绍	42
4.12.2. 监控服务的插件	43
4.12.3. 创建一个主机对象定义	43
4.12.4. 创建服务对象定义	44
4.12.5. 监控HTTP	44
4.12.6. 监控FTP	45
4.12.7. 监控SSH	46
4.12.8. 监控SMTP	47
4.12.9. 监控POP3	47
4.12.10. 监控IMAP	48
4.12.11. 重启动Nagios	49
第 5 章 准备配置Nagios	49

5.1. 配置概览	49
5.1.1. 介绍	49
5.1.2. 主配置文件	50
5.1.3. 资源配置文件	50
5.1.4. 对象定义文件	50
5.1.5. CGI配置文件	50
5.2. 主配置文件选项	50
5.2.1. 配置文件的位置	51
5.2.2. 配置文件里的变量	51
5.3. 对象配置概览	92
5.3.1. 什么是对象?	92
5.3.2. 对象在哪里定义?	93
5.3.3. 对象如何定义?	93
5.4. CGI配置文件选项	95
5.4.1. 样例配置文件	95
5.4.2. 配置文件的位置	95
5.4.3. 配置文件里的变量	95
第 6 章 Nagios监控与配置的基本概念	103
6.1. 对象定义	103
6.1.1. 介绍	103
6.1.2. 注意状态保持设置	103
6.1.3. 样例配置文件	104
6.1.4. 对象种类	104
6.1.5. 主机定义	104
6.1.6. 主机组定义	113
6.1.7. 服务定义	115
6.1.8. 服务组定义	122
6.1.9. 联系人定义	123
6.1.10. 联系人组定义	127
6.1.11. 时间周期定义	128
6.1.12. 命令定义	132
6.1.13. 服务依赖定义	133
6.1.14. 服务扩展定义	136
6.1.15. 主机依赖定义	138
6.1.16. 主机扩展定义	141
6.1.17. 额外主机信息定义	143
6.1.18. 额外服务信息定义	146
6.2. 对象定义的省时诀窍	148
6.2.1. 介绍	148
6.2.2. 正则式匹配	149
6.2.3. 服务的定义	149
6.2.4. 服务扩展的定义	150
6.2.5. 服务依赖的定义	152
6.2.6. 主机扩展的定义	155

6.2.7. 主机依赖的定义	156
6.2.8. 主机组的定义	157
6.3. 用户自定义对象变量	157
6.3.1. 介绍	157
6.3.2. 用户自定义变量的基本规则	157
6.3.3. 例子	158
6.3.4. 在宏里使用用户自定义变量	158
6.3.5. 用户自定义变量与继承	159
6.4. 对象继承关系	159
6.4.1. 介绍	159
6.4.2. 基础	159
6.4.3. 本地变量和继承变量比较	160
6.4.4. 继承关系链	161
6.4.5. 用不完整的对象定义做模板	162
6.4.6. 用户定义变量	164
6.4.7. 取消继承的字串值	164
6.4.8. 继承时附加字串值	165
6.4.9. 隐含继承	166
6.4.10. 在对象扩展里的隐含与附加继承	167
6.4.11. 多重继承	167
6.4.12. 在多重继承中指定优先级	169
6.5. 计划停机时间	170
6.5.1. 介绍	170
6.5.2. 计划停机时间	170
6.5.3. 固定的与可变的停机时间	170
6.5.4. 触发停机时间	171
6.5.5. 计划停机时间对通知产生什么影响?	171
6.5.6. 计划停机时间的重叠	171
6.6. 时间周期	172
6.6.1. 介绍	172
6.6.2. 时间周期中的优先权	172
6.6.3. 时间周期在主机与服务检测时是如何起作用的?	172
6.6.4. 时间周期在联系人通知时是如何起作用的?	173
6.6.5. 时间周期在通知扩展里是如何起作用的?	173
6.6.6. 时间周期在依赖关系里是如何起作用的?	173
6.7. 通知	173
6.7.1. 介绍	173
6.7.2. 何时会做通知?	174
6.7.3. 谁会收到通知?	174
6.7.4. 送出通知时必须要通过什么样的过滤器?	174
6.7.4.1. 程序层面的过滤器	174
6.7.4.2. 主机与服务过滤器	174
6.7.4.3. 联系人过滤器	175
6.7.5. 通知的方式	175

6.7.6. 通知类型的宏	176
6.7.7. 有用的资源	177
6.8. 事件处理	177
6.8.1. 介绍	177
6.8.2. 何时执行事件处理?	178
6.8.3. 事件处理类型	178
6.8.4. 使能事件处理	179
6.8.5. 事件处理的执行次序	179
6.8.6. 编写事件处理命令	179
6.8.7. 事件处理命令的权限	179
6.8.8. 服务事件处理的例子	179
6.9. 外部命令	183
6.9.1. 介绍	183
6.9.2. 使能外部命令	183
6.9.3. Nagios什么时候用外部命令检测?	183
6.9.4. 使用外部命令	183
6.9.5. 命令格式	184
6.10. 状态类型	184
6.10.1. 介绍	184
6.10.2. 服务与主机的检测重试	184
6.10.3. 软态	184
6.10.4. 硬态	185
6.10.5. 举例	185
6.11. 主机检测	186
6.11.1. 介绍	186
6.11.2. 什么时候做主机检测?	186
6.11.3. 缓存主机检测	187
6.11.4. 依赖性与检测	187
6.11.5. 并发主机检测	187
6.11.6. 主机状态	188
6.11.7. 主机状态判定	188
6.11.8. 主机状态变换	189
6.12. 服务检测	189
6.12.1. 介绍	189
6.12.2. 什么时候会做服务检测?	189
6.12.3. 缓存服务检测	189
6.12.4. 依赖性与检测	189
6.12.5. 服务检测并发	190
6.12.6. 服务状态	190
6.12.7. 服务状态判定	190
6.12.8. 服务状态变换	190
6.13. 自主检测	190
6.13.1. 介绍	190
6.13.2. 自主检测是如何进行的?	190

6.13.3. 什么时间执行自主检测?	191
6.14. 强制检测	191
6.14.1. 介绍	191
6.14.2. 强制检测的用处	191
6.14.3. 强制检测是如何工作的?	192
6.14.4. 使能强制检测	192
6.14.5. 提交服务的强制检测结果	192
6.14.6. 提交主机的强制检测结果	193
6.14.7. 强制检测与主机状态	193
6.14.8. 判定来自远程主机的强制检测结果.....	194
第 7 章 运行Nagios的基本操作	194
7.1. 验证配置文件的正确性	194
7.2. 启动与停止Nagios	195
7.2.1. 启动Nagios	195
7.2.2. 重新启动Nagios	195
7.2.3. 停止Nagios	195
7.3. 快速启动选项	196
7.3.1. 介绍	196
7.3.2. 背景	196
7.3.3. 评估启动时间	196
7.3.4. 预缓存对象配置	198
7.3.5. 跳过回路检测	199
7.3.6. 联合起来使用	200
7.4. 关于CGI程序模块的信息	200
7.4.1. 说明	200
7.4.2. 索引	200
第 8 章 Nagios深入进阶	209
8.1. Nagios的插件	209
8.1.1. 介绍	209
8.1.2. 什么是插件?	209
8.1.3. 插件是一个抽象层	209
8.1.4. 什么样的插件可用?	210
8.1.5. 获得插件	210
8.1.6. 如何来使用插件X?	210
8.1.7. 插件API	210
8.2. 理解Nagios宏及其工作机制	210
8.2.1. 宏	210
8.2.2. 宏替换 — 宏的工作机制	210
8.2.3. 例 1: 主机IP地址宏	211
8.2.4. 例 2: 命令参数宏	211
8.2.5. 按需而成的宏 (on-demand macro).....	212
8.2.6. 用户自定制宏	213
8.2.7. 宏的清理	213

8.2.8. 作为环境变量的宏	214
8.2.9. 可用宏	214
8.3. Nagiosr内嵌的标准宏	214
8.3.1. 宏的有效性	214
8.3.2. 可利用的宏图表	215
8.3.3. 宏的描述说明	224
8.3.4. 注意	241
8.4. 如何确认网络中主机的状态与可达性	241
8.4.1. 介绍	241
8.4.2. 样板网络	242
8.4.3. 定义网络主机的父子关系	242
8.4.4. 可达性逻辑的运转	244
8.4.5. 不可达状态与通知	246
8.5. 可变服务	246
8.5.1. 介绍	246
8.5.2. 可用于什么情况?	247
8.5.3. 可变服务有何特别之处?	247
8.5.4. The Power Of Two	247
8.6. 主机与服务的刷新检测	248
8.6.1. 介绍	249
8.6.2. 刷新检测如何工作?	249
8.6.3. 使能刷新检测	249
8.6.4. 样例	250
8.7. 感知和处理状态抖动	251
8.7.1. 介绍	251
8.7.2. 感知抖动是如何工作的?	251
8.7.3. 例子	252
8.7.4. 服务的抖动感知	253
8.7.5. 主机的抖动感知	253
8.7.6. 抖动检测门限	253
8.7.7. 给抖动检测所用的状态	254
8.7.8. 抖动处理	254
8.7.9. 使能抖动感知功能	254
8.8. 服务和主机的定期检测	255
8.8.1. 未完成	255
8.9. 有关通知的对象扩展	255
8.9.1. 介绍	255
8.9.2. 什么时候做通知扩展?	255
8.9.3. 联系人组	256
8.9.4. 扩展范围的覆盖	257
8.9.5. 恢复的通知	258
8.9.6. 通知间隔	258
8.9.7. 时间周期的限制	261

8.9.8. 状态限制	261
8.10. 应召循环	261
8.10.1. 介绍	261
8.10.2. 场景 1: 假日与周末	262
8.10.3. 场景 2: 轮换值班	263
8.10.4. 场景 3: 轮换周值班	264
8.10.5. 场景 4: 假期	266
8.10.6. 其他场景	268
8.11. 主机间与服务间依赖关系	268
8.11.1. 介绍	268
8.11.2. 服务依赖概况	268
8.11.3. 定义服务依赖	268
8.11.4. 服务依赖对象的样例	268
8.11.5. 如何测试服务依赖?	271
8.11.6. 实施依赖	271
8.11.7. 通知依赖	272
8.11.8. 依赖关系的继承	272
8.11.9. 主机依赖	273
8.11.10. 主机依赖关系的样例	273
8.12. 依赖检测的前处理	274
8.12.1. 介绍	274
8.12.2. 如何进行依赖检测前准备工作?	274
8.12.3. 使能检查准备	275
8.12.4. 缓存检测	275
8.13. 性能数据	275
8.13.1. 介绍	276
8.13.2. 性能数据的类型	276
8.13.3. 插件返回的性能数据	276
8.13.4. 性能数据的处理	277
8.13.5. 用命令来处理性能数据	277
8.13.6. 将性能数据写入文件	278
第 9 章 Nagios 专业话题	278
9.1. 趣事与玩笑	278
9.2. 分布式监控	278
9.2.1. 介绍	278
9.2.2. 目标	278
9.2.3. 参照示意图	278
9.2.4. 中心服务与分布服务的对比	279
9.2.5. 从分布服务器上收集服务检测信息	280
9.2.6. 分布式监控服务的配置	280
9.2.7. 中心服务器配置	283
9.2.8. 强制检测中的问题	283
9.2.9. 刷新检测	283

9.2.10. 执行主机检测	285
9.3. 冗余式与失效式网络监控	285
9.3.1. 介绍	285
9.3.2. 索引	286
9.4. 大型安装模式的变化	292
9.4.1. 介绍	292
9.4.2. 影响	292
9.5. 缓存检测	293
9.5.1. 介绍	293
9.5.2. 只为按需检测使用	293
9.5.3. 缓存检测是如何工作的?	294
9.5.4. 这将到底意味着什么?	294
9.5.5. 配置变量参数	295
9.5.6. 优化缓存效率	295
9.6. 状态追踪	296
9.6.1. 介绍	296
9.6.2. 它是如何工作的?	297
9.6.3. 需要使能状态追踪么?	298
9.6.4. 如何使能追踪?	298
9.6.5. 状态追踪与可变服务有何不同?	298
9.6.6. 限制与告诫	298
9.7. 集群主机和集群服务的监控	298
9.7.1. 介绍	298
9.7.2. Plan of Attack	299
9.7.3. 使用集群检测check_cluster插件.....	299
9.7.4. 监控服务集群	299
9.7.5. 主机集群的监控	300
9.8. 适应性监控	301
9.8.1. 介绍	301
9.8.2. 什么可以改?	301
9.8.3. 适应性监控的外部命令	302
9.9. 强制式主机状态迁移	302
9.9.1. 介绍	302
9.9.2. 不同的全局视图	302
9.9.3. 使能状态迁移	303
第 10 章 Nagios自身的安全性 with 性能调优	304
10.1. 自身安全相关事项	304
10.1.1. 介绍	304
10.1.2. Best Practices	305
10.2. Nagios的性能调优	307
10.2.1. 介绍	307
10.2.2. 优化的招数:	308

10.3. 使用Nagios状态工具	312
10.3.1. 介绍	312
10.3.2. 用法信息	312
10.3.3. 可阅读的输出	312
10.3.4. MRTG集成	314
10.4. 使用MRTG绘制性能数据	314
10.4.1. 介绍	314
10.4.2. MRTG配置样例	314
10.4.3. 图表实例	315
10.5. 对CGIs程序模块的授权与认证	318
10.5.1. 介绍	318
10.5.2. 定义	318
10.5.3. 设置认证用户	318
10.5.4. 打开CGI模块的认证与授权功能	319
10.5.5. 给CGI模块的默认许可权限	319
10.5.6. 给CGI增加额外的权限	320
10.5.7. CGI模块的授权要求	321
10.5.8. 在加密的Web服务器上认证	321
10.6. 用户定制CGI页面头和尾	321
10.6.1. 介绍	321
10.6.2. 它是如何工作的?	322
第 11 章 软件集成相关的内容	322
11.1. 软件集成概览	322
11.1.1. 介绍	322
11.1.2. 集成的要点	323
11.1.3. 集成事例	323
11.2. SNMP陷阱集成	323
11.2.1. 介绍	323
11.3. TCP Wrapper集成	323
11.3.1. 介绍	324
11.3.2. 定义一个服务	324
11.3.3. 配置TCP Wrappers	325
11.3.4. 写那个脚本	325
11.3.5. 搞好了	326
11.4. Nagios外部构件	326
11.4.1. 介绍	326
11.4.2. NRPE	326
11.4.3. NSCA	326
11.4.4. NDOUtils	327
第 12 章 开发相关	327
12.1. 使用内嵌Perl解释器	327
12.1.1. 介绍	327

12.1.2. 优点	328
12.1.3. 缺点	328
12.1.4. 使用嵌入式Perl解释器	329
12.1.5. 编译一个Nagios带嵌入式Perl解释器.....	329
12.1.6. 指定插件使用Perl解释器	329
12.1.7. 开发嵌入式Perl解释器可运行的Perl插件.....	330
12.2. 开发使用内嵌式Perl解释器的Nagios插件	330
12.2.1. 介绍	330
12.2.2. 目标受众	330
12.2.3. 开发Perl插件必备知识	330
12.2.4. 开发ePN下的Perl插件必做内容	331
12.3. Nagios插件API	334
12.3.1. 其他资源	334
12.3.2. 插件概览	334
12.3.3. 返回值	334
12.3.4. 特定插件输出	335
12.3.5. 插件输出样例	335
12.3.6. 插件输出长度限制	337
12.3.7. 例子	337
12.3.8. Perl插件	337

第 2 章 关于 Nagios

2.1. 什么是 Nagios?

Nagios 是一款用于系统和网络监控的应用程序。它可以在你设定的条件下对主机和服务进行监控，在状态变差和变好的时候给出告警信息。

Nagios最初被设计为在 [Linux](#)系统之上运行，然而它同样可以在类Unix的系统之上运行。

Nagios 更进一步的特征包括：

- 监控网络服务（SMTP、POP3、HTTP、NNTP、PING 等）；
- 监控主机资源（处理器负荷、磁盘利用率等）；
- 简单地插件设计使得用户可以方便地扩展自己服务的检测方法；
- 并行服务检查机制；
- 具备定义网络分层结构的能力，用“parent”主机定义来表达网络主机间的关系，这种关系可被用来发现和明晰主机宕机或不可达状态；
- 当服务或主机问题产生与解决时将告警发送给联系人（通过 EMail、短信、用户定义方式）；
- 具备定义事件句柄功能，它可以在主机或服务的事件发生时获取更多问题定位；
- 自动的日志回滚；
- 可以支持并实现对主机的冗余监控；
- 可选的 WEB 界面用于查看当前的网络状态、通知和故障历史、日志文件等；

2.2. 系统需求

Nagios 所需要的运行条件是机器必须可以运行 Linux（或是 Unix 变种）并且有 C 语言编译器。你必须正确地配置 TCP/IP 协议栈以使大多数的服务检测可以通过网络得以进行。

你需要**但并非必须**正确地配置 Nagios 里的 CGI 程序，而一旦你要使用 CGI 程序时，你必须安装以下这些软件...

- 一个WEB服务（最好是 [Apache](#)）
- Thomas Boutell制作的 [gd库](#)版本应是 1.6.3 或更高（在CGIs程序模块 [statusmap](#)和 [trends](#)这两个模块里需要这个库）

2.3. 版权

Nagios版权遵从于由 [自由软件基金会](#)所发布的GNU版权协议第二版。有关GNU协议请查阅 [自由软件基金会](#)网站。该版权协议允许你在某些条件下可以复制、分发并且或者是修改它。可以在Nagios软件发行包里阅读版权文件LICENSE或是在网站上阅读 [在线版权](#)文件以获取更多信息。

Nagios is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE WARRANTY OF DESIGN, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

2.4. 致谢

一些人对Nagios的发布尽力，不管是报告错误、提供建议、编写插件等等，可以在网站 <http://www.nagios.org/>上找到这些人的名字列表。

2.5. 下载最新版本

可以在Nagios<http://www.nagios.org/>站点获取最新版本。

注意

*Nagios 及 Nagios 商业标识由 Ethan Galstad 所拥有。*其他的商业标识、服务标识、注册商标及注册服务属于各自的所有者。

第 3 章 Nagios 3.0 新特性

第 3 章 Nagios 3.0 新特性

重要

Important: Make sure you read through the documentation and the FAQs at <http://www.nagios.org/> before sending a question to the mailing lists.

3.1. 更新日志

Nagios的更新日志可以在这里的 [在线文件](#)或是在源程序的发行包的根目录里找到。

3.2. 变更与新特征

- 文档:
 - 更新了文档 — 很抱歉我对文档的更新工作进展迟缓。这会花些时间来做，因为有很多文档而且写这些文档并不是我喜欢的事情（我更不喜欢整天翻译，这也不是我喜欢的事情）。期待一些文档与其他的有所不同，而这些不同会对于那些新人或有经验的 Nagios 使用者起些作用。
- 内嵌宏:
 - 新加宏 — 加入了一些新宏，包括：\$TEMPPATH\$、\$LONGHOSTOUTPUT\$、\$LONGSERVICEOUTPUT\$、\$HOSTNOTIFICATIONID\$、\$SERVICENOTIFICATIONID\$、\$HOSTEVENTID\$、\$SERVICEEVENTID\$、

\$SERVICEISVOLATILE\$、\$LASTHOSTEVENTID\$、\$LASTSERVICEEVENTID\$、\$HOSTDISPLAYNAME\$、
\$SERVICEDISPLAYNAME\$、\$MAXHOSTATTEMPTS\$、\$MAXSERVICEATTEMPTS\$、
\$TOTALHOSTSERVICES\$、\$TOTALHOSTSERVICESOK\$、\$TOTALHOSTSERVICESWARNING\$、
\$TOTALHOSTSERVICESUNKNOWN\$、\$TOTALHOSTSERVICESCRITICAL\$、\$CONTACTGROUPNAME\$、
\$CONTACTGROUPNAMES\$、\$CONTACTGROUPALIAS\$、\$CONTACTGROUPMEMBERS\$、
\$NOTIFICATIONRECIPIENTS\$、\$NOTIFICATIONISESCALATED\$、\$NOTIFICATIONAUTHOR\$、
\$NOTIFICATIONAUTHORNAME\$、\$NOTIFICATIONAUTHORALIAS\$、\$NOTIFICATIONCOMMENT\$、
\$EVENTSTARTTIME\$、\$HOSTPROBLEMID\$、\$LASTHOSTPROBLEMID\$、\$SERVICEPROBLEMID\$、
\$LASTSERVICEPROBLEMID\$、\$LASTHOSTSTATE\$、\$LASTHOSTSTATEID\$、\$LASTSERVICESTATE\$、
\$LASTSERVICESTATEID\$。加入了两个特殊的守护时间宏：\$ISVALIDTIME:\$和
\$NEXTVALIDTIME:\$。

- 移除的宏 — 原有的宏\$NOTIFICATIONNUMBER\$被分离为两个新宏
\$HOSTNOTIFICATIONNUMBER\$和\$SERVICENOTIFICATIONNUMBER\$。
- 变更的宏 — 现有的\$HOSTNOTES\$和\$SERVICENOTES\$宏包括自身外，还包括
\$HOSTNOTESURL\$、\$HOSTACTIONURL\$、\$SERVICENOTESURL\$和\$SERVICEACTIONURL\$等几个宏。
- 在检测、事件句柄处理、告警和其他外部命令执行时，宏可以获取环境变量。这可能会使Nagios
在大型部署方案时占用较高的CPU处理能力，你可以设置 [enable_environment_macros](#) 选
项来不使能它。
- 有关宏的更新信息可以在 [这里](#) 查到。
- 预定义停机时间：
 - [预定义停机时间](#)不再保存在各自文件(之前是由主配置文件里的downtime_file来指定)。
当前的和保留的预定义停机时间将分别保存于 [状态文件](#)和 [保留文件retention file](#)中。
- 注释：
 - 主机和服务的注释不再保存于各自的文件(之前在主配置文件中的comment_file来指定)。
当前的和保留的注释将分别保存于 [状态文件status file](#)和 [保留文件retention file](#)之中。
 - Acknowledgement comments that are marked as non-persistent are now only deleted
when the acknowledgement is removed. They were previously automatically deleted
when Nagios restarted, which was not ideal.
- State Retention Data:
 - Status information for individual contacts is now retained across program restarts.

- Comment and downtime IDs are now retained across program restarts and should be unique unless the retention data is deleted or ignored.
- Added [retained_host_attribute_mask](#) and [retained_service_attribute_mask](#) variables to control what host/service attributes are retained globally across program restarts.
- Added [retained_process_host_attribute_mask](#) and [retained_process_service_attribute_mask](#) variables to control what process attributes are retained across program restarts.
- Added [retained_contact_host_attribute_mask](#) and [retained_contact_service_attribute_mask](#) variables to control what contact attributes are retained globally across program restarts.
- Flap Detection:
 - Added **flap_detection_options** directive to host and service definitions to allow you to specify what host/service states should be used by the flap detection logic (by default all states are used).
 - Percent state change and state history are now retained and recorded even when flap detection is disabled.
 - Hosts and services are immediately checked for flapping when flap detection is enabled program-wide.
 - Hosts and services that are flapping when flap detection is disabled program-wide are now logged.
 - More information on flap detection can be found [here](#).
- External Commands:
 - Added a new `PROCESS_FILE` external command to allow processing of external commands found in an external (regular) file. Useful for processing large amounts of passive checks with long output, or for scripting regular commands. More information can be found [here](#).
 - Custom commands may now be submitted to Nagios. Custom command names are prefixed with an underscore and are not processed internally by the Nagios daemon. They may, however, be processed by a loaded NEB module.

- The [check_external_commands](#) option is now enabled by default, which means Nagios is configured to check for external "commands out of the box". All 2.x and earlier versions of Nagios had this option disabled by default.
- Status Data:
 - Contact status information (last notification times, notifications enabled/disabled, etc.) is now saved in the [status](#) and [retention](#) files, although it is not processed by the CGIs.
- Embedded Perl:
 - Added new [enable_embedded_perl](#) and [use_embedded_perl_implicitly](#) variables to control use of the embedded Perl interpreter.
 - Perl scripts/plugins can now explicitly tell Nagios whether or not they should be run under the embedded Perl interpreter. This is useful if you have troublesome scripts that don't function well under the ePN.
 - More information about these new options can be found [here](#).
- Adaptive Monitoring:
 - The check timeperiod for hosts and services can now be modified on-the-fly with the appropriate external command (CHANGE_HOST_CHECK_TIMEPERIOD or CHANGE_SVC_CHECK_TIMEPERIOD). 查阅这个 [网页](#) 以取得更多可用的适应性检测命令。
- Notifications:
 - A **first_notification_delay** option has been added to host and service definitions to (what else) introduce a delay between when a host/service problem first occurs and when the first problem notification goes out. In previous versions you had to use some mighty config-fu with escalations to accomplish this. Now this feature is available to normal mortals.
 - Notifications are now sent out for hosts/services that are flapping when flap detection is disabled on a host- or service-specific basis or on a program-wide basis. The \$NOTIFICATIONTYPE\$ macro will be set to "FLAPPINGDISABLED" in this situation.
 - Notifications can now be sent out when scheduled downtime start, ends, and is cancelled for hosts and services. The \$NOTIFICATIONTYPE\$ macro will be set to "DOWNTIMESTART", "DOWNTIMEEND", or "DOWNTIMECANCELLED", respectively. In order to

received notifications on scheduled downtime events, specify "s" or "downtime" in your contact, host, and/or service notification options.

- More information on notifications can be found [here](#).
- Object Definitions:
 - Service dependencies can now be created to easily define "same host" dependencies for different services on one or more hosts. ([Read more](#))
 - Extended host and service definitions (hostextinfo and serviceextinfo, respectively) have been deprecated. All values that from extended definitions have been merged with host or service definitions, as appropriate. Nagios 3 will continue to read and process older extended information definitions, but will log a warning. Future versions of Nagios (4.x and later) will not support separate extended info definitions.
 - New **hostgroup_members**, **servicegroup_members**, and **contactgroup_members** directives have been added to hostgroup, servicegroup, and contactgroups definitions, respectively. This allows you to include hosts, services, or contacts from sub-groups in your group definitions.
 - New **notes**, **notes_url**, and **action_url** have been added to hostgroup and servicegroup definition.
 - Contact definitions have the new **host_notifications_enabled**, **service_notifications_enabled**, and **can_submit_commands** directives to better control notifications and determine whether or not they can submit commands through the web interface.
 - Host and service dependencies now support an optional **dependency_period** directive. This allows you to limit the times during which dependencies are valid.
 - The **parallelize** directive in service definitions is now deprecated and no longer used. All service checks are run in parallel in Nagios 3.
 - There are no longer any inherent limitations on the length of host names or service descriptions.
 - Extended regular expressions are now used if you enable the [use_regexp_matching](#) config option. Regular expression matching is only used in certain object definition directives that contain *, ?, +, or \.

- A new **initial_state** directive has been added to host and service definitions, so you can tell Nagios that a host/service should default to a specific state when Nagios starts, rather than UP or OK (which is still the default).
- Object Inheritance:
 - You can now inherit object variables/values from multiple templates by specifying more than one template name in the **use** directive of object definitions. This can allow for some very powerful (and complex) inheritance setups. ([Read more](#))
 - Services now inherit contact groups, notification interval, and notification period from their associated host if not otherwise specified. ([Read more](#))
 - Host and service escalations now inherit contact groups, notification interval, and escalation timeperiod from their associated host or service if not otherwise specified. ([Read more](#))
 - String variables in host, service, and contact definitions can now be prevented from being inherited by specifying a value of "null" (without quotes) for the value of the variable. ([Read more](#))
 - Most string variables in local object definitions can now be appended to the string values that are inherited. This is quite handy in large configurations. ([Read more](#))
- Performance Improvements:
 - Add ability to precache object config files and exclude circular path detection checks from verification process. This can speed up Nagios start time immensely in large environments! Read more [here](#).
 - A new [use large installation tweaks](#) option has been added that should improve performance in large Nagios installations. Read more about this [here](#).
 - A number of internal improvements have been made with regards to how Nagios deals with internal data structures and object (e.g. host and service) relationships. These improvements should result in a speedup for larger installations.
 - New [external command buffer slots](#) option has been added to allow you to more easily scale Nagios in large environments. For best results you should consider using [MRTG to graph](#) Nagios' usage of buffer slots over time.
- Plugin Output:

- Multiline plugin output is now supported for host and service checks. Hooray! The plugin API has been updated to support multiple lines of output in a manner that retains backward compatability with older plugins. Additional lines of output (aside from the first line) are now stored in new `$LONGHOSTOUTPUT$` and `$LONGSERVICEOUTPUT$` macros.
- The maximum length of plugin output has been increased to 4K (from around 350 bytes in previous versions). This 4K limit has been arbitrarily chosen to protect against runaway plugins that dump back too much data to Nagios.
- More information on the plugins, multiline output, and max plugin output length can be found [here](#).
- Service Checks:
 - Nagios now checks for orphaned service checks by default.
 - Added a new [enable predictive service dependency checks](#) option to control whether or not Nagios will initiate predictive check of service that are being depended upon (in dependency definitions). Predictive checks help ensure that the dependency logic is as accurate as possible. ([Read more](#))
 - A new cached service check feature has been implemented that can significantly improve performance for many people. Instead of executing a plugin to check the status of a service, Nagios can often use a cached service check result instead. More information on this can be found [here](#).
- Host Checks:
 - Host checks are now run in parallel! Host checks used to be run in a serial fashion, which meant they were a major holdup in terms of performance. No longer! ([Read more](#))
 - Host check retries are now performed like service check retries. That is to say, host definitions now have a new **retry_interval** that specifies how much time to wait before trying the host check again. :-)
 - Regularly scheduled host checks now longer hinder performance. In fact, they can help to increase performance with the new cached check logic (see below).
 - Added a new [check_for_orphaned_hosts](#) option to enable checks of orphaned host checks. This is need now that host checks are run in parallel.

- Added a new [enable_predictive_host_dependency_checks](#) option to control whether or not Nagios will initiate predictive check of hosts that are being depended upon (in dependency definitions). Predictive checks help ensure that the dependency logic is as accurate as possible. ([Read more](#))
- A new cached host check feature has been implemented that can significantly improve performance for many people. Instead of executing a plugin to check the status of a host, Nagios can often use a cached host check result instead. More information on this can be found [here](#).
- Passive host checks that have a DOWN or UNREACHABLE result can now be automatically translated to their proper state from the point of view of the Nagios instance that receives them. This is very useful in failover and distributed monitoring setups. More information on passive host check state translation can be found [here](#).
- Passive host checks normally put a host into a HARD state. This can now be changed by enabling the [passive_host_checks_are_soft](#) option.
- Freshness checks:
 - A new [additional_freshness_latency](#) option has been added to allow you specify the number of seconds that should be added to any host or service freshness threshold that is automatically calculated by Nagios.
- IPC:
 - The IPC mechanism that is used to transfer host/service check results back to the Nagios daemon from (grand)child processes has changed! This should help to reduce load/latency issues related to processing large numbers of passive checks in distributed monitoring environments.
 - Check results are now transferred by writing check results to files in directory specified by the [check_result_path](#) option. Files that are older than the [max_check_result_file_age](#) option will be mercilessly deleted without further processing.
- Timeperiods:
 - Timeperiods were overdue for a major overhaul and have finally been extended to allow for date exceptions, skip dates (every 3 days), etc! This should help you out when defining notification timeperiods for pager rotations.

- More information on the new timeperiod directives can be found [here](#) and [here](#).
- Event Broker:
 - Updated NEB API version
 - Modified callback for adaptive program status data
 - Added callback for adaptive contact status data
 - Added precheck callbacks for hosts and services to allow modules to cancel/override internal host/service checks.
- Web Interface:
 - [enable splunk integrations](#)
 - Hostgroup and servicegroup summaries now show important/unimportant problem breakdowns like the TAC CGI.
 - Minor layout changes to host and service detail views in extinfo CGI.
 - New check statistics and have been added to the "Performance Info" screen.
 - Added [Splunk](#)
 - Added new [notes_url_target](#) and [action_url_target](#) options to control what frame notes and action URLs are opened in.
 - Added new [lock_author_names](#) option to prevent alteration of author names when users submit comments, acknowledgements, and scheduled downtime.
- Debugging Info:
 - The DEBUGx compile options available in the configure script for have been removed.
 - Debugging information can now be written to a separate debug file, which is automatically rotated when it reaches a user-defined size. This should make debugging problems much easier, as you don't need to recompile Nagios. Full support for writing debugging information to file is being added during the alpha development phase, so it may not be complete when you try it.
 - Variables that affect the debug log in [debug file](#), [debug level](#), [debug verbosity](#), and [max debug file size](#).
- Misc:
 - Temp path variable – A new [temp_path](#) variable has been added to specify a scratch directory that Nagios can use for temporary scratch space.

- Unique notification and event ID numbers – A unique ID number is now assigned to each host and service notification. Another unique ID is now assigned to all host and service state changes as well. The unique IDs can be accessed using the following respective macros: `$HOSTNOTIFICATIONID$`, `$SERVICENOTIFICATIONID$`, `$HOSTEVENTID$`, `$SERVICEEVENTID$`, `$LASTHOSTEVENTID$`, `$LASTSERVICEEVENTID$`.
- New macros – A few new macros (other than those already mentioned elsewhere above) have been added. They include `$HOSTGROUPNAMES$`, `$SERVICEGROUPNAMES$`, `$HOSTACKAUTHORNAME$`, `$HOSTACKAUTHORALIAS$`, `$SERVICEACKAUTHORNAME$`, and `$SERVICEACKAUTHORALIAS$`.
- Reaper frequency – The old **`service_reaper_frequency`** variable has been renamed to [`check_result_reaper_frequency`](#), as it is now also used to process host check results.
- Max reaper time – A new [`max_check_result_reaper_time`](#) variable has been added to limit the amount of time a single reaper event is allowed to run.
- Fractional intervals – Fractional notification and check intervals (e.g. "3.5" minutes) are now supported in host, service, host escalation, and service escalation definitions.
- Escaped command arguments – You can now pass bang (!) characters in your command arguments by escaping them with a backslash (\). If you need to include backslashes in your command arguments, they should also be escaped with a backslash.
- Multiline system command output – Nagios will now read multiple lines out output from system commands it runs (notification scripts, etc.), up to 4K. This matches the limits on plugin output mentioned earlier. Output from system commands is not directly processed by Nagios, but support for it is there nonetheless.
- Better scheduling information – More detailed information is given when Nagios is executed with the `-s` command line option. This information can be used to help [reduce](#) the time it takes to start/restart Nagios.
- Aggregated status file updates – The old **`aggregate_status_updates`** option has been removed. All status file updates are now aggregated at a minimum interval of 1 second.

- New performance data file mode – A new “p” option has been added to the [host_perfdata_file_mode](#) and [service_perfdata_file_mode](#) options. This new mode will open the file in non-blocking read/write mode, which is useful for pipes.
- Timezone offset – A new [use_timezone](#) option has been added to allow you to run different instances of Nagios in timezones different from the local zone.

第 4 章 入门

第 4 章 入门

4.1. 给新手的建议

祝贺你选择了 Nagios! Nagios 是一个非常强大且柔性化的软件, 但可能需要不少心血来学习如何配置使之符合你所需, 一旦掌握了它如何工作并怎样来工作时, 你会觉得再也离不开它! :-) 对于初次使用 Nagios 的新手这有几个建议需要遵从:

- **放松点 — 这会花些时间。**不要指望它事情可以在转瞬间就搞掂, 没有那么容易。设置好 Nagios 是一个费点事的工作, 部分是由于对 Nagios 设置并不清楚, 而还可能是由于并不清楚如何来监控现有网络 (或者说如何监控会更好)。
- **使用快速上手指南。**本帮助给出了 [快速安装指南](#)是给那些新手尽快地将Nagios安装到位并运行起来而写就的。在不到二十分钟之内可以安装并监控本地的系统, 一旦完成了, 就可以继续学习配置Nagios了。
- **阅读文档。**如果掌握 Nagios 运行机制, 可以高效地配置它并且使之无所不能。确信已经阅读了这些文档(是“配置 Nagios”和“基本操作”两章)。在更好地理解基础性配置之前可以对那些高级内容暂时不管。
- **获得他人协助。**如果已经阅读文档并检测了样本配置文件但仍然有问题, 写一个EMail给 [nagios-users](#)邮件列表并写清楚问题。由于在这个项目上我有不少事情要做, 直接给我的邮件我可能无法回复, 所以最好是求助于邮件列表, 如果有较好的背景并且可以将问题描述清楚, 或许有人可以指出如何正确来做。更多地信息请在这个链接 <http://www.nagios.org/support/>下寻找。

4.2. 旧 Nagios 升级到当前版本

目录

- [第 4.2.1 节 “从旧的 3.x版本升级到当前版本”](#)
- [第 4.2.2 节 “从 2.x升级到 3.x”](#)
- [第 4.2.3 节 “从RPM包安装状态升级”](#)

4.2.1. 从旧的 3.x 版本升级到当前版本

如果是使用 3.x 的旧版，肯定是要尽快升级到当前版本。新版本修正了许多错误，下面假定已经根据[快速安装指南](#)的操作步骤从源代码包开始安装好 Nagios，下面可以安装更新的版本。虽然下面的操作都是用 root 操作的，但可以不用 root 权限也可以升级成功。下面是升级过程...

先确认已经备份好现有版本的 Nagios 软件和配置文件。如果升级过程中有不对的，至少可以回退到旧版本。

切换为 Nagios 用户。使用 Debian/Ubuntu 系统的可以用 `sudo -s nagios` 来切换。

```
su -l nagios
```

下载最新的 Nagios 安装包 (<http://www.nagios.org/download/>)。

```
wget http://osdn.dl.sourceforge.net/sourceforge/nagios/nagios-3.x.tar.gz
```

展开源码包。

```
tar xzf nagios-3.x.tar.gz
```

```
cd nagios-3.x
```

运行 Nagios 源程序的配置脚本，把加入外部命令的组名加上，象这样：

```
./configure --with-command-group=nagcmd
```

编译源程序

```
make all
```

安装升级后的二进制程序、文档和 Web 接口程序。在这步时旧配置文件还不会被覆盖。

```
make install
```

验证配置并重新启动 Nagios

```
/usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
```

```
/sbin/service nagios restart
```

好了，升级完成！

4.2.2. 从 2.x 升级到 3.x

Nagios 从 2.x 升级到 3.x 并不难。升级过程如同上面的旧版 3.x 的升级过程。但是 Nagios 3.x 中有几处配置文件的改动需要注意：

- 主配置文件里的原 `service_reaper_frequency` 变量已经更名为 [check_result_reaper_frequency](#);

- 原来的宏`$NOTIFICATIONNUMBER$`已经被拆分为新宏 [\\$HOSTNOTIFICATIONNUMBER\\$](#)和 [\\$SERVICENOTIFICATIONNUMBER\\$](#)。
- 原来服务对象定义里的 `parallelize` 选项不推荐用，以后不再使用，全部的服务检测将并行化运行；
- 移除了原有的 `aggregate_status_updates` 选项，全部的状态文件的更新都是以最短 1 秒的间隔来同时进行的；
- 主机扩展和服务扩展对象定义不推荐用，Nagios 仍旧读入并处理它，但推荐的是把这些主机和服务的域设置分别转到在主机和服务的对象定义里；
- 在主配置文件里的`downtime_file`文件变量将不再支持，取而代之的是把计划停机时间全部保存在 [状态保留文件](#)里。为维持再有停机设置，停下Nagios 2.x程序，并把旧的停机时间文件里的内容加到状态保留文件里；
- 在主配置文件里的`comment_file`文件变量将不再支持，取而代之的是把注释全部保存在 [状态保留文件](#)里。为留下再有的注释，可停下Nagios 2.x程序并把再有注释内容全部加到状态保留文件里。

还要保证阅读过本文档的“[新特性](#)”一节的内容。它给出了 3.x新版本的与最终稳定的 2.x版本间的变化，变化有不少需要好好读一下。

4.2.3. 从 RPM 包安装状态升级

如果当前是用 RPM 包安装的，或是用 Debian/Ubuntu 的 APT 软件包来安装 Nagios 的，需要用源程序包来安装升级，下面是操作步骤：

- 主配置文件(一般是 `nagios.cfg`)
- 资源文件(一般是 `resource.cfg`)
- CGI 配置文件(一般是 `cgi.cfg`)
- 全部的对象定义文件
- 配置文件
- 状态保留文件(一般是 `retention.dat`)
- 当前的 Nagios 日志文件(一般是 `nagios.log`)
- 打包的 Nagios 日志文件
- 备份现有 Nagios 安装程序里的：
- 卸载原始的 Nagios 的 RPM 包或是 APT 包；
- 按照 [快速安装指南](#)完成自源程序安装Nagios的工作；
- 恢复原始的 Nagios 配置文件，状态保留文件和日志文件；
- [验证](#)配置文件并 [重新启动](#)Nagios。

注意 RPM 和 APT 包把 Nagios 的文件放置的位置有所不同。在升级前要确保那些配置文件备份好以在碰到解决不了的升级问题时可以回退到旧版本。

4.3. 快速安装指南

4.3.1. 介绍

这些指南试图让你在二十分钟内用简单地指令操作下从源程序安装 Nagios 并监控你的本地机器。这里并不讨论那些高级指令对于 95%以上的想起步的用户而言这是基础。

4.3.2. 指南

现在可以提供如下 Linux 发行版本上的快速安装指南：

- [基于Fedora平台的快速指南](#)
- [基于openSUSE平台的快速指南](#)
- [基于Ubuntu平台的快速指南](#)

你可以在 NagiosCommunity.org的维基百科上找到更多的安装上手指南。什么？找不到你所用的操作系统版本的指南？在维基百科上给其他人写一条吧！

如果你在一个上面没列出的操作系统或Linux发行包上安装Nagios，请参照 [Fedora快速指南](#)来概要地了解一下你需要做的事情。命令名、路径等可能因不同的发行包或操作系统而不同，因而这时你可能需要些努力来搞一下安装文档里的东西。

4.3.3. 安装后该做的

一旦你正确地安装并使 Nagios 运行起来后，毫无疑问你不仅要监控你的主机，你需要审视一下更多的文档来做更多的事情...

- [监控Windows主机](#)
- [监控Linux/Unix主机](#)
- [监控Netware服务器](#)
- [监控路由器和交换机](#)
- [监控网络打印机](#)
- [监控公众服务平台](#)

4.4. 基于 Fedora 平台的快速指南

4.4.1. 介绍

本指南试图让你通过简单的指令以在 20 分钟内在 Fedora 平台上通过对 Nagios 的源程序的安装来监控本地主机。这里没有讨论更高级的设置项 — 只是一些基本操作，但这足以使 95%的用户启动 Nagios。

这些指令在基于 Fedora Core 6 的系统下写成的。

最终结果是什么

如果按照本指南安装，最后将是这样结果：

- Nagios 和插件将安装到/usr/local/nagios
- Nagios 将被配置为监控本地系统的几个主要服务(CPU 负荷、磁盘利用率等)
- Nagios 的 Web 接口是 URL 是 http://localhost/nagios/

4.4.2. 准备软件包

在做安装之前确认要对该机器拥有 root 权限。

确认你安装好的 Fedora 系统上已经安装如下软件包再继续。

- Apache
- GCC 编译器
- [GD](#)库与开发库

可以用 **yum** 命令来安装这些软件包，键入命令：

```
yum install httpd
yum install gcc
yum install glibc glibc-common
yum install gd gd-devel
```

4.4.3. 操作过程

1) 建立一个帐号

切换为 root 用户

```
su -l
```

创建一个名为 **nagios** 的帐号并给定登录口令

```
/usr/sbin/useradd nagios
passwd nagios
```

创建一个用户组名为 **nagcmd** 用于从 Web 接口执行外部命令。将 nagios 用户和 apache 用户都加到这个组中。

```
/usr/sbin/groupadd nagcmd
/usr/sbin/usermod -G nagcmd nagios
/usr/sbin/usermod -G nagcmd apache
```

2) 下载 Nagios 和插件程序包

建立一个目录用以存储下载文件

```
mkdir ~/downloads  
cd ~/downloads
```

下载Nagios和Nagios插件的软件包(访问 <http://www.nagios.org/download/> 站点以获得最新版本), 在写本文档时, 最新的Nagios的软件版本是 3.0rc1, Nagios插件的版本是 1.4.11。

```
wget http://osdn.dl.sourceforge.net/sourceforge/nagios/nagios-3.0rc1.tar.gz  
wget http://osdn.dl.sourceforge.net/sourceforge/nagiosplug/nagios-plugins-1.4.11.tar.gz
```

3) 编译与安装 Nagios

展开 Nagios 源程序包

```
cd ~/downloads  
tar xzf nagios-3.0rc1.tar.gz  
cd nagios-3.0rc1
```

运行 Nagios 配置脚本并使用先前开设的用户及用户组:

```
./configure --with-command-group=nagcmd
```

编译 Nagios 程序包源码

```
make all
```

安装二进制运行程序、初始化脚本、配置文件样本并设置运行目录权限

```
make install  
make install-init  
make install-config  
make install-commandmode
```

现在还不能启动 Nagios—还有一些要做的...

4) 客户化配置

样例 [配置文件](#)默认安装在这个目录下 **/usr/local/nagios/etc**, 这些样例文件可以配置Nagios使之正常运行, 只需要做一个简单的修改...

用你擅长的编辑器软件来编辑这个 **/usr/local/nagios/etc/objects/contacts.cfg** 配置文件, 更改 email 地址 **nagiosadmin** 的联系人定义信息中的 EMail 信息为你的 EMail 信息以接收报警内容。

```
vi /usr/local/nagios/etc/objects/contacts.cfg
```

5) 配置 WEB 接口

安装 Nagios 的 WEB 配置文件到 Apache 的 conf.d 目录下

```
make install-webconf
```

创建一个 **nagiosadmin** 的用户用于 Nagios 的 WEB 接口登录。记下你所设置的登录口令，一会儿你会用到它。

```
htpasswd -c /usr/local/nagios/etc/htpasswd.users nagiosadmin
```

重启 Apache 服务以使设置生效。

```
service httpd restart
```

6) 编译并安装 Nagios 插件

展开 Nagios 插件的源程序包

```
cd ~/downloads
tar xzf nagios-plugins-1.4.11.tar.gz
cd nagios-plugins-1.4.11
```

编译并安装插件

```
./configure --with-nagios-user=nagios --with-nagios-group=nagios
make
make install
```

7) 启动 Nagios

把 Nagios 加入到服务列表中以使之在系统启动时自动启动

```
chkconfig --add nagios
chkconfig nagios on
```

验证 Nagios 的样例配置文件

```
/usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
```

如果没有报错，可以启动 Nagios 服务

```
service nagios start
```

8) 更改 SELinux 设置

Fedora 与 SELinux(安全增强型 Linux)同步发行与安装后将默认使用强制模式。这会在你尝试联入 Nagios 的 CGI 时导致一个“内部服务错误”消息。

如果是 SELinux 处于强制安全模式时需要做

```
getenforce
```

令 SELinux 处于容许模式

```
setenforce 0
```

如果要永久性更变它，需要更改/etc/selinux/config 里的设置并重启系统。

不关闭 SELinux 或是永久性变更它的方法是让 CGI 模块在 SELinux 下指定强制目标模式：

```
chcon -R -t httpd_sys_content_t /usr/local/nagios/sbin/
```

```
chcon -R -t httpd_sys_content_t /usr/local/nagios/share/
```

更多有关Nagios的CGI模块增加目标策略的强制权限方式见NagiosCommunity.org的维基百科

<http://www.nagioscommunity.org/wiki>。

9) 登录 WEB 接口

你现在可以从 WEB 方式来接入 Nagios 的 WEB 接口了，你需要在提示下输入你的用户名(nagiosadmin)和口令，你刚刚设置的，这里用系统默认安装的浏览器，用下面这个超链接

```
http://localhost/nagios/
```

点击“服务详情”的引导超链来查看你本机的监视详情。你可能需要给点时间让 Nagios 来检测你机器上所依赖的服务因为检测需要些时间。

10) 其他的变更

确信你机器的防火墙规则配置允许你可以从远程登录到 Nagios 的 WEB 服务。

配置EMail的报警项超出了本文档的内容，指向你的系统档案用网页查找或是到这个站点

[NagiosCommunity.org wiki](http://www.nagioscommunity.org/wiki)来查找更进一步的信息，以使你的系统上可以向外部地址发送EMail信息。更多有关通知的信息可以查阅 [这篇](#)文档。

11) 完成了

祝贺你已经成功安装好 Nagios，但网络监控工作只是刚开始。毫无疑问你不是只监控本地系统，所以要看以下这些文档...

- [对Windows主机的监控](#)
- [对Linux/Unix主机的监控](#)
- [对Netware服务器的监控](#)
- [监控路由器和交换机](#)
- [监控公众化服务](#) (HTTP、FTP、SSH等)

4.5. 基于 openSUSE 平台的快速指南

4.5.1. 介绍

本指南试图让你通过简单的指令以在 20 分钟内在你的 openSUSE 平台上通过对 Nagios 的源程序的安装来监控本地主机。这里没有讨论更高级的设置项 — 只是一些基本操作，但这足以使 95% 的用户启动 Nagios。

这些指令在基于 openSUSE10.2 的系统下写成的。

4.5.2. 所需的软件包

确认你安装好的 openSUSE 系统之上已经安装了如下软件包再继续。你可以在 openSUSE 系统下用 **yast** 来安装软件包。

- apache2
- C/C++ 开发库

4.5.3. 操作过程

1) 建立一个帐号

切换为 root 用户

```
su -l
```

创建新帐户名为 **nagios** 并给它一个登录口令

```
/usr/sbin/useradd nagios  
passwd nagios
```

创建一个用户组名为 **nagios**，并把 nagios 帐户加入该组

```
/usr/sbin/groupadd nagios  
/usr/sbin/usermod -G nagios nagios
```

创建一个用户组名为 **nagcmd** 来执行外部命令并可以通过 WEB 接口来执行。将 nagios 用户和 apache 用户都加到这个组中。

```
/usr/sbin/groupadd nagcmd  
/usr/sbin/usermod -G nagcmd nagios  
/usr/sbin/usermod -G nagcmd wwwrun
```

2) 下载 Nagios 和插件程序包

建立一个目录用以存储下载文件

```
mkdir ~/downloads
```



```
cd ~/downloads
```

下载Nagios和Nagios插件的软件包（访问 <http://www.nagios.org/download/> 站点以获得最新版本），在写本文档时，最新的Nagios的软件版本是 3.0rc1，Nagios插件的版本是 1.4.11。

```
wget http://osdn.dl.sourceforge.net/sourceforge/nagios/nagios-3.0rc1.tar.gz
wget http://osdn.dl.sourceforge.net/sourceforge/nagiosplug/nagios-plugins-1.4.11.tar.gz
```

3) 编译与安装 Nagios

展开 Nagios 源程序包

```
cd ~/downloads
tar xzf nagios-3.0rc1.tar.gz
cd nagios-3.0rc1
```

运行 Nagios 配置脚本并使用先前开设的用户及用户组：

```
./configure --with-command-group=nagcmd
```

编译 Nagios 程序包源码

```
make all
```

安装二进制运行程序、初始化脚本、配置文件样本并设置运行目录权限

```
make install
make install-init
make install-config
make install-commandmode
```

现在还不能启动 Nagios — 还有一些要做的...

4) 客户化配置

样例 [配置文件](#)默认安装在这个目录下 `/usr/local/nagios/etc`，这些样例文件可以配置Nagios使之正常运行，只需要做一个简单的修改...

用你擅长的编辑器软件来编辑这个 `/usr/local/nagios/etc/objects/contacts.cfg` 配置文件，更改 email 地址 `nagiosadmin` 的联系人定义信息中的 EMail 信息为你的 EMail 信息以接收报警内容。

```
vi /usr/local/nagios/etc/objects/contacts.cfg
```

5) 配置 WEB 接口

安装 Nagios 的 WEB 配置文件到 Apache 的 `conf.d` 目录下

```
make install-webconf
```

创建一个 **nagiosadmin** 的用户用于 Nagios 的 WEB 接口登录。记下你所设置的登录口令，一会儿你会用到它。

```
htpasswd2 -c /usr/local/nagios/etc/htpasswd.users nagiosadmin
```

重启 Apache 服务以使设置生效。

```
service apache2 restart
```

6) 编译并安装 Nagios 插件

展开 Nagios 插件的源程序包

```
cd ~/downloads
tar xzf nagios-plugins-1.4.11.tar.gz
cd nagios-plugins-1.4.11
```

编译并安装插件

```
./configure --with-nagios-user=nagios --with-nagios-group=nagios
make
make install
```

7) 启动 Nagios

把 Nagios 加入到服务列表中以使之在系统启动时自动启动

```
chkconfig --add nagios
chkconfig nagios on
```

验证 Nagios 的样例配置文件

```
/usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
```

如果没有报错，可以启动 Nagios 服务

```
service nagios start
```

8) 登录 WEB 接口

你现在可以从 WEB 方式来接入 Nagios 的 WEB 接口了，你需要在提示下输入你的用户名(**nagiosadmin**)和口令，你刚刚设置的，这里用系统默认安装的浏览器，用下面这个超链接

```
konqueror http://localhost/nagios/
```

点击“服务详情”的引导超链来查看你本机的监视详情。你可能需要给点时间让 Nagios 来检测你机器上所依赖的服务因为检测需要些时间。

9) 其他的变更

确信你机器的防火墙规则配置允许你可以从远程登录到 Nagios 的 WEB 服务。

你可以这样做：

- 打开控制中心
- 选择‘打开超户设置’以打开 YaST 超户控制中心
- 选择在‘安全与用户’设置里的‘防火墙’
- 在防火墙的配置窗口中点击‘允许的服务’选项
- 在许可的服务中增加‘HTTP 服务’，是‘外部区’的部分
- 点击‘下一步’并选择‘接受’以使得防火墙设置生效

配置EMail的报警项超出了本文档的内容，指向你的系统档案用网页查找或是到这个站点

[NagiosCommunity.org wiki](http://NagiosCommunity.org/wiki)来查找更进一步的信息，以使你的openSUSE系统上可以向外部地址发送EMail信息。

4.6. 基于 Ubuntu 平台的快速指南

4.6.1. 介绍

本指南试图让你通过简单的指令以在 20 分钟内在 Ubuntu 平台上通过对 Nagios 的源程序的安装来监控本地主机。没有讨论更高级的设置项—只是一些基本操作，但这足以使 95%的用户启动 Nagios。

这些指令在基于 Ubuntu6.10(桌面版)的系统下写成的。

What You'll End Up With

如果按照本指南安装，最后将是这样结果：

- Nagios 和插件将安装到/usr/local/nagios
- Nagios 将被配置为监控本地系统的几个主要服务(CPU 负荷、磁盘利用率等)
- Nagios 的 Web 接口是 URL 是 <http://localhost/nagios/>

4.6.2. 所需软件包

确认你安装好的系统上已经安装如下软件包再继续。

- Apache2
- GCC 编译器与开发库
- GD 库与开发库

可以用 **apt-get** 命令来安装这些软件包，键入命令：

```
sudo apt-get install apache2
```

```
sudo apt-get install build-essential  
sudo apt-get install libgd2-dev
```

4.6.3. 操作过程

1) 建立一个帐号

切换为 root 用户

```
sudo -s
```

创建一个名为 **nagios** 的帐号并给定登录口令

```
/usr/sbin/useradd nagios  
passwd nagios
```

在 Ubuntu 服务器版 (6.01 或更高版本), 创建一个用户组名为 **nagios** (默认是不创建的)。在 Ubuntu 桌面版上要跳过这一步。

```
/usr/sbin/groupadd nagios  
/usr/sbin/usermod -G nagios nagios
```

创建一个用户组名为 **nagcmd** 用于从 Web 接口执行外部命令。将 nagios 用户和 apache 用户都加到这个组中。

```
/usr/sbin/groupadd nagcmd  
/usr/sbin/usermod -G nagcmd nagios  
/usr/sbin/usermod -G nagcmd www-data
```

2) 下载 Nagios 和插件程序包

建立一个目录用以存储下载文件

```
mkdir ~/downloads  
cd ~/downloads
```

下载 Nagios 和 Nagios 插件的软件包 (访问 <http://www.nagios.org/download/> 站点以获得最新版本), 在写本文档时, 最新的 Nagios 的软件版本是 3.0rc1, Nagios 插件的版本是 1.4.11。

```
wget http://osdn.dl.sourceforge.net/sourceforge/nagios/nagios-3.0rc1.tar.gz  
wget http://osdn.dl.sourceforge.net/sourceforge/nagiosplug/nagios-plugins-1.4.11.tar.gz
```

3) 编译与安装 Nagios

展开 Nagios 源程序包

```
cd ~/downloads
tar xzf nagios-3.0rc1.tar.gz
cd nagios-3.0rc1
```

运行 Nagios 配置脚本并使用先前开设的用户及用户组：

```
./configure --with-command-group=nagcmd
```

编译 Nagios 程序包源码

```
make all
```

安装二进制运行程序、初始化脚本、配置文件样本并设置运行目录权限

```
make install
make install-init
make install-config
make install-commandmode
```

现在还不能启动 Nagios—还有一些要做的...

4) 客户化配置

样例 [配置文件](#)默认安装在这个目录下 **/usr/local/nagios/etc**，这些样例文件可以配置 Nagios 使之正常运行，只需要做一个简单的修改...

用你擅长的编辑器软件来编辑这个 **/usr/local/nagios/etc/objects/contacts.cfg** 配置文件，更改 email 地址 **nagiosadmin** 的联系人定义信息中的 EMail 信息为你的 EMail 信息以接收报警内容。

```
vi /usr/local/nagios/etc/objects/contacts.cfg
```

5) 配置 WEB 接口

安装 Nagios 的 WEB 配置文件到 Apache 的 conf.d 目录下

```
make install-webconf
```

创建一个 **nagiosadmin** 的用户用于 Nagios 的 WEB 接口登录。记下你所设置的登录口令，一会儿你会用到它。

```
htpasswd -c /usr/local/nagios/etc/htpasswd.users nagiosadmin
```

重启 Apache 服务以使设置生效。

```
/etc/init.d/apache2 reload
```

6) 编译并安装 Nagios 插件

展开 Nagios 插件的源程序包

```
cd ~/downloads
tar xzf nagios-plugins-1.4.11.tar.gz
cd nagios-plugins-1.4.11
```

编译并安装插件

```
./configure --with-nagios-user=nagios --with-nagios-group=nagios
make
make install
```

7) 启动 Nagios

把 Nagios 加入到服务列表中以使之在系统启动时自动启动

```
ln -s /etc/init.d/nagios /etc/rcS.d/S99nagios
```

验证 Nagios 的样例配置文件

```
/usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
```

如果没有报错，可以启动 Nagios 服务

```
/etc/init.d/nagios start
```

8) 登录 WEB 接口

你现在可以从 WEB 方式来接入 Nagios 的 WEB 接口了，你需要在提示下输入你的用户名(nagiosadmin)和口令，你刚刚设置的，这里用系统默认安装的浏览器，用下面这个超链接

```
http://localhost/nagios/
```

点击“服务详情”的引导超链来查看你本机的监视详情。你可能需要给点时间让 Nagios 来检测你机器上所依赖的服务因为检测需要些时间。

9) 其他的变更

如果要接收 Nagios 的 EMail 警报，需要安装(Postfix)包

```
sudo apt-get install mailx
```

需要编辑 Nagios 里的 EMail 通知送出命令，它位于/usr/local/nagios/etc/commands.cfg 文件中，将里面的'/bin/mail'全部替换为'/usr/bin/mail'。一旦设置好需要重启动 Nagios 以使配置生效。

```
sudo /etc/init.d/nagios restart
```

配置E-Mail的报警项超出了本文档的内容，指向你的系统档案用网页查找或是到这个站点

NagiosCommunity.org/wiki来查找更进一步的信息，以使Ubuntu系统上可以向外部地址发送E-Mail信息。

4.7. 监控 Windows 主机

4.7.1. 介绍

本文用来说明如何监控 Windows 主机的本地服务和特性，包括：

- 内存占用率
- CPU 负载
- Disk 利用率
- 服务状态
- 运行进程
- 等等

在Windows主机上的公众化服务(如HTTP、FTP、POP3 等)可以查阅 [监控公众化服务](#)这篇文档。

注意



如下的内容是假定你已经按照 [快速安装指南](#)安装好了Nagios系统之后做的，下面所使用的样例配置文件(如**commands.cfg**、**templates.cfg**等)已经在安装过程中安装到位。

4.7.2. 概览

对 Windows 机器的监控私有服务需要在机器上安装代理程序。代理将会在检测插件与 Nagios 服务之间起网关代理作用。如果没有在机器上安装代理的话，Nagios 将无法对 Windows 私有服务或属性等进行监控。

在下面例子中，将在Windows机器上安装 [NSClient++](#)外部构件并使用**check_nt**插件检测和与NSClient++构件进行通讯。如果你按照指南来安装的话，**check_nt**插件已经安装到了Nagios服务器上。

如果愿意，可以用其他的Windows代理(象 [NC Net](#))替代NSClient++构件所起的作用—只是要稍稍改一下对应的命令和服务定义等。下面将只是讨论安装了NSClient++外部构件的情况。

4.7.3. 步骤

为完成对 Windows 机器的检测，有几个步骤要做，它们是：

- 确认一下首要条件；
- 在 Windows 机器上安装代理(在本例中是安装 NSClient++构件)；
- 给 Windows 机器创建新的主机和服务对象定义；

- 重新启动 Nagios 守护进程。

4.7.4. 已经做了什么？

为使过程简单，已经完成了少量配置文件的工作：

- 已经把 **check_nt** 命令加入到了 **commands.cfg** 文件中，就可以直接使用 **check_nt** 插件来监控 Windows 服务；
- 一个 Windows 机器的主机对象模板(命名为 **windows-server**)已经在 **templates.cfg** 文件里创建好了，可以更容易地加入一个新的 Windows 主机对象定义。

常用的配置文件可以在 **/usr/local/nagios/etc/objects/** 目录里找到。如果愿意可以对里面的对象进行修改以适应你的要求。但是，如果你没有熟悉配置 Nagios 之前劝你不要这么做。开始时可以只是按照下面的指令操作来快速完成监控 Windows 机器。

4.7.5. 首要条件

首次监控一台 Windows 机器时需要对 Nagios 做点额外的工作，记住，仅仅是监控第一台 Windows 机器时需要做这些工作。

编辑 Nagios 的主配置文件

```
vi /usr/local/nagios/etc/nagios.cfg
```

把下面这行最前面的#号去掉：

```
#cfg_file=/usr/local/nagios/etc/objects/windows.cfg
```

保存配置文件并退出。

刚才做的是什么呢？是让 Nagios 起用 **/usr/local/nagios/etc/objects/windows.cfg** 这个配置文件里的对象定义。在这个配置文件里可以加些 Windows 的主机与服务对象定义。该配置文件里已经包含有几个样例主机、主机组及服务对象定义。对于第一台 Windows 机器，可以只是简单地修改里面已经有的主机与服务对象定义而不要新创建一个。

4.7.6. 安装 Windows 代理程序

在用 Nagios 监控 Windows 机器的私有服务之前，需要先在机器上安装代理程序。推荐使用 NSClient++ 外部构件，它可以在 <http://sourceforge.net/projects/nscplus> 找到。如下指令可以安装一个基本的 NSClient++ 外部构件，同时也配置好 Nagios 来监控这台 Windows 机器。

1. 从 <http://sourceforge.net/projects/nscplus> 站点下载最新稳定版的 NSClient++ 软件包；
2. 展开软件包到一个目录下，如 C:\NSClient++；
3. 打开一个命令行窗口并切换到 C:\NSClient++ 目录下；
4. 用下面命令将 NSClient++ 系统服务注册到系统里：


```
nsclient++ /install
```

5. 用下面命令安装 NSClient++ 系统托盘程序('SysTray' 是大小写敏感的):

```
nsclient++ SysTray
```

6. 打开服务管理器并确认 NSClientpp 服务可以在桌面交互(看一下服务管理器里的'Log On' 选项页), 如果没有允许桌面交互, 点一下里面的选择项打开它。



7. 编辑 NSC. INI 文件(位于 C:\NSClient++目录)并做如下修改:

- 去掉在[modules]段里的列出模块程序的注释, 除了 CheckWMI.dll 和 RemoteConfiguration.dll;
- 最好是修改一下在[Settings]段里的'password' 选项;
- 去掉在[Settings]段里的'allowed_hosts' 选项注释,把 Nagios 服务所在主机的 IP 加到这一行里, 或是置为空, 让全部主机都可以联入;
- 确认一下在[NSClient]段里的'port' 选项里已经去掉注释并设置成'12489' (默认端口);

8. 用下面命令启动 NSClient++服务:

```
nsclient++ /start
```

9. 如果安装正确, 一个新的图标会出现在系统托盘里, 是个黄圈里面有个黑色的'M';

10. 完成了。这台 Windows 机器可以加到 Nagios 监控配置里了...

4.7.7. 配置 Nagios

为监控Windows机器下面要在Nagios配置文件里加几个 [对象定义](#)。

编辑方式打开 **windows.cfg** 文件。

```
vi /usr/local/nagios/etc/objects/windows.cfg
```

给Windows机器加一个新的 [主机对象](#) 定义以便监控。如果是被监控的第一台Windows机器，可以只是修改 **windows.cfg** 文件里的对象定义。修改 **host_name**、**alias** 和 **address** 域以符合那台Windows机器。

```
define host{
    use                windows-server    ; Inherit default values from a Windows server
    template (make sure you keep this line!)
    host_name          winserver
    alias              My Windows Server
    address             192.168.1.2
}
```

好了。下面可以加几个服务定义(在同一个配置文件里)以使 Nagios 监控 Windows 机器上的不同属性内容。如果是第一台 Windows 机器，可以只是修改 **windows.cfg** 里的服务对象定义。

注意



用你刚刚加好的主机对象定义里的 **host_name** 来替换例子里的“**winserver**”。

加入下面的服务定义以监控运行于 Windows 机器上的 NSClient++ 外部构件的版本。当到时间要升级 Windows 机器上的外部构件时这信息会很用有，因为它可以告知这台 Windows 机器上的 NSClient++ 需要升级到最新版本。

```
define service{
    use                generic-service
    host_name          winserver
    service_description NSClient++ Version
    check_command       check_nt!CLIENTVERSION
}
```

加入下面的服务定义以监控 Windows 机器的启动后运行时间。

```
define service{
    use                generic-service
    host_name          winserver
    service_description Uptime
```

```
        check_command          check_nt!UPTIME
    }
```

加入下面的服务定义可监控 Windows 机器的 CPU 利用率, 并在 5 分钟 CPU 负荷高于 90%时给出一个紧急警报或是高于 80%时给出一个告警警报。

```
define service{
    use                generic-service
    host_name          winserver
    service_description CPU Load
    check_command      check_nt!CPULOAD!-l 5,80,90
}
```

加入下面的服务定义可监控 Windows 机器的内存占用率, 并在 5 分钟内存占用率高于 90%时给出一个紧急警报或是高于 80%时给出一个告警警报。

```
define service{
    use                generic-service
    host_name          winserver
    service_description Memory Usage
    check_command      check_nt!MEMUSE!-w 80 -c 90
}
```

加入下面的服务定义可监控 Windows 机器的 C: 盘的磁盘利用率, 并在磁盘利用率高于 90%时给出一个紧急警报或是高于 80%时给出一个告警警报。

```
define service{
    use                generic-service
    host_name          winserver
    service_description C:\ Drive Space
    check_command      check_nt!USEDISKSPACE!-l c -w 80 -c 90
}
```

加入下面的服务定义可监控 Windows 机器上的 W3SVC 服务状态, 并在 W3SVC 服务停止时给出一个紧急警报。

```
define service{
```

```

use                generic-service

host_name          winserver

service_description W3SVC

check_command      check_nt!SERVICESTATE!-d SHOWALL -l W3SVC
}

```

加入下面的服务定义可监控 Windows 机器上的 Explorer.exe 进程，并在进程没有运行时给出一个紧急警报。

```

define service{

    use                generic-service

    host_name          winserver

    service_description Explorer

    check_command      check_nt!PROCSTATE!-d SHOWALL -l Explorer.exe
}

```

都好了，已经加好了基础服务定义，可以监控 Windows 机器了，保存一下配置文件。

4.7.8. 口令保护

如果想指定保存在 Windows 机器上 NSClient++ 配置文件里的口令，可以修改 **check_nt** 命令定义，让它带着口令。编辑方式打开 **commands.cfg** 文件。

```
vi /usr/local/nagios/etc/commands.cfg
```

修改 **check_nt** 命令的定义，带上“-s <PASSWORD>”命令参数(这里的 PASSWORD 要换成你 Windows 机器的真正口令)，象这样：

```

define command{

    command_name      check_nt

    command_line      $USER1$/check_nt -H $HOSTADDRESS$ -p 12489 -s PASSWORD -v $ARG1$
$ARG2$

}

```

保存文件退出。

4.7.9. 重启 Nagios

如果修改好 Nagios 配置文件，需要 [验证你的配置文件](#) 并 [重启 Nagios](#)。

如果验证配置文件过程中有什么错误信息，在做下一步前一定要修正好配置文件。一定要保证验证过程中不再有出错信息后再启动或重新启动 Nagios！

4.8. 监控 Linux/Unix 主机

4.8.1. 介绍

本文档描述了如果监控 Linux/UNIX 的“私有”服务和属性，如：

- CPU 负荷
- 内存占用率
- 磁盘利用率
- 登录用户
- 运行进程
- 等

由Linux系统上的公众服务(HTTP、FTP、SSH、SMTP等)可以按照这篇 [监控公众服务](#) 文档。

注意



如下内容是假定已经按照 [快速安装指南](#) 安装并设置好 Nagios。如下例子参考了样例配置文件 (`commands.cfg`、`templates.cfg`等)里的对象定义，样例配置文件已经在安装过程中安装就位。

4.8.2. 概览

[注意：本文档没有结束。推荐阅读文档 [NRPE外部构件](#)里如何监控远程Linux/Unix服务器中的指令]

有几种不同方式来监控远程 Linux/UNIX 服务器的服务与属性。一个是应用共享式 SSH 密钥运行 `check_by_ssh` 插件来执行对远程主机的检测。这种方法本文档不讨论，但它会导致安装有 Nagios 的监控服务器很高的系统负荷，尤其是你要监控成百个主机中的上千个服务时，这是因为要建立/毁构 SSH 联接的总开销很高。

另一种方法是使用 [NRPE外部构件](#)监控远程主机。NRPE外部构件可以在远程的Linux/Unix主机上执行插件程序。如果是要象监控本地主机一样对远程主机的磁盘利用率、CPU负荷和内存占用率等情况下，NRPE 外部构件非常有用。

4.9. 监控路由器和交换机

4.9.1. 介绍

本文档将介绍如何来监控路由器和交换机的状态。一些便宜的“无网管”功能的交换机与集线器不能配置 IP 地址而且对于网络是不可见的组成构件，因而没办法来监控这种东西。稍贵些的交换机和路由器可以配置 IP 地址可以用 PING 检测或是通过 SNMP 来查询状态信息。

下面将描述如果来监控这些有网管功能的交换机、集线器和路由器：

- 包丢弃率，平均回包周期 RTA
- SNMP 状态信息
- 带宽与流量

注意



如下指令是假定你已经按 [快速安装指南](#) 安装好 Nagios。参考的样例配置是在已经按指南安装就位的配置文件 (`commands.cfg`、`templates.cfg` 等)。

4.9.2. 概览

监控交换机与路由器可简可繁——主要是看拥有什么样设备与想监控什么内容。做为极为重要的网络组成构件，毫无疑问至少要监控一些基本状态。

交换机与路由器可以简单地用 PING 来监控丢包率、RTA 等数据。如果交换机支持 SNMP，就可以监控端口状态等，用 `check_snmp` 插件，也可以监控带宽 (如果用了 MRTG)，用 `check_mrtgtraf` 插件。

`check_snmp` 插件只有当系统里安装了 `net-snmp` 和 `net-snmp-utils` 包后才编译。先确定插件已经在 `/usr/local/nagios/libexec` 目录里再继续做，如果没有这个文件，安装 `net-snmp` 和 `net-snmp-utils` 包并且重编译并重新安装 Nagios 插件包。

4.9.3. 步骤

要监控交换机与路由器要有几步工作：

- 第一时间执行些必备工作；
- 给设备创建要监控的主机与服务对象定义；
- 重启动 Nagios 守护进程。

4.9.4. 已经做了什么？

为了让工作轻松点，几个配置任务已经做好了：

- 两个命令定义 (`check_snmp` 和 `check_local_mrtgtraf`) 已经加到了 `commands.cfg` 文件中。可以用 `check_snmp` 和 `check_mrtgtraf` 插件来监控网络打印机。
- 一个交换机模板 (命名为 `generic-switch`) 已经创建在 `templates.cfg` 文件里。可以在对象定义里更容易地加一个新的交换机与路由器设备。

以上的监控配置文件可以在 `/usr/local/nagios/etc/objects/` 目录里找到。如果愿意可以修改这些定义或是加入其他适合需要的更好的定义。但推荐你最好是等到你熟练地掌握了 Nagios 配置之后再这么做。开始的时候，只要按上述的配置来监控网络里的路由器和交换机就可以了。

4.9.5. 必备工作

要配置 Nagios 用于监控网络里的交换机之前，有必要做点额外工作。记住，这是首先要做的工作才能监控。

编辑 Nagios 的主配置文件

```
vi /usr/local/nagios/etc/nagios.cfg
```

移除文件里下面这行的最前面的(＃)符号

```
#cfg_file=/usr/local/nagios/etc/objects/switch.cfg
```

保存文件并退出。

为何要这么做？这是要让 Nagios 检查 `/usr/local/nagios/etc/objects/switch.cfg` 配置文件来找些额外的对象定义。在文件里可以增加有关路由器和交换机设备的主机与服务定义。配置文件已经包含了几个样本主机、主机组和服务定义。做为监控路由器与交换机的第一步工作是最好在样例的主机与服务对象定义之上修改而不是重建一个。

4.9.6. 配置 Nagios

需要做些 [对象定义](#) 以监控新的交换机与路由器设备。

打开 `switch.cfg` 文件进行编辑。

```
vi /usr/local/nagios/etc/objects/switch.cfg
```

给要监控的交换机加一个新的 [主机](#) 对象定义。如果这是第一台要监控的交换机设备，可以简单地修改 `switch.cfg` 里的样例配置。修改主机对象里的 `host_name`、`alias` 和 `address` 域值来适用于监控。

```
define host{
    use                generic-switch        ; Inherit default values from a template
    host_name          linksys-srw224p       ; The name we're giving to this switch
    alias              Linksys SRW224P Switch ; A longer name associated with the switch
    address            192.168.1.253         ; IP address of the switch
    hostgroups         allhosts,switches ; Host groups this switch is associated with
}
```

4.9.7. 监控服务

现在可以加些针对监控交换机的服务对象定义(在同一个配置文件)。如果是第一台要监控的交换机设备，可以简单地修改 `switch.cfg` 里的样例配置。

注意



替换样例定义里的“linksys-srw224p”主机名为你刚才定义的名字，是修改在 **host_name** 域。

4.9.8. 监控丢包率和 RTA

增加如下的服务定义以监控自 Nagios 监控主机到交换机的丢包率和平均回包周期 RTA，在一般情况下每 5 分钟检测一次。

```
define service{
    use                generic-service    ; Inherit values from a template
    host_name          linksys-srw224p    ; The name of the host the service is
associated with
    service_description PING              ; The service description
    check_command       check_ping!200.0,20%!600.0,60%    ; The command used to
monitor the service
    normal_check_interval 5              ; Check the service every 5 minutes under normal
conditions
    retry_check_interval 1               ; Re-check the service every minute until its
final/hard state is determined
}
```

这个服务的状态将会处于：

- 紧急 (CRITICAL) — 条件是 RTA 大于 600ms 或丢包率大于等于 60%；
- 告警 (WARNING) — 条件是 RTA 大于 200ms 或是丢包率大于等于 20%；
- 正常 (OK) — 条件是 RTA 小于 200ms 或丢包率小于 20%

4.9.9. 监控 SNMP 状态信息

如果交换机与路由器支持 SNMP 接口，可以用 **check_snmp** 插件来监控更丰富的信息。如果不支持 SNMP，跳过此节。

加入如下服务定义到你刚才修改的交换机对象定义之中

```
define service{
    use                generic-service    ; Inherit values from a template
    host_name          linksys-srw224p
    service_description Uptime
    check_command       check_snmp!-C public -o sysUpTime.0
}
```



```
}
```

在上述服务定义中的 **check_command** 域里, 用“-C public”来指定 SNMP 共同体名称为“public”, 用“-o sysUpTime.0”指明要检测的 OID(译者注—MIB 节点值)。

如果要确保交换机上某个指定端口或接口的状态处于运行状态, 可以在对象定义里加入一段定义:

```
define service{
    use                generic-service    ; Inherit values from a template
    host_name          linksys-srw224p
    service_description Port 1 Link Status
    check_command       check_snmp!-C public -o ifOperStatus.1 -r 1 -m RFC1213-MIB
}
```

在上例中, “-o ifOperStatus.1”指出取出交换机的端口编号为 1 的 OID 状态。“-r 1”选项是让 **check_snmp** 插件检查返回一个正常 (OK) 状态, 如果是在 SNMP 查询结果中存在“1”(1 说明交换机端口处于运行状态) 如果没找到 1 就是紧急 (CRITICAL) 状态。“-m RFC1213-MIB”是可选的, 它告诉 **check_snmp** 插件只加载“RFC1213-MIB”库而不是加载每个在系统里的 MIB 库, 这可以加快插件运行速度。

这就是给 SNMP 库的例子。有成百上千种信息可以通过 SNMP 来监控, 这完全取决于你需要做什么和如果来做监控。祝你好运!

提示



通常可以用如下命令来寻找你想用于监控的 OID 节点(用你的交换机 IP 替换 192.168.1.253): **snmpwalk -v1 -c public 192.168.1.253 -m ALL .1**

4.9.10. 监控带宽和流量

可以监控交换机或路由器的带宽利用率, 用 [MRTG](#) 绘图并让 Nagios 在流量超出指定门限时报警。

check_mrtgtraf 插件(它已经包含在 Nagios 插件软件发行包中)可以实现。

需要让 **check_mrtgtraf** 插件知道如何来保存 MRTG 数据并存入文件, 以及门限等。在例子中, 监控了一个 Linksys 交换机。MRTG 日志保存于 **/var/lib/mrtg/192.168.1.253_1.log** 文件中。这就是我用于监控的服务定义, 它可以用于监控带宽数据到日志文件之中...

```
define service{
    use                generic-service    ; Inherit values from a template
    host_name          linksys-srw224p
```

```
service_description      Port 1 Bandwidth Usage

check_command

check_local_mrtgtraf!/var/lib/mrtg/192.168.1.253_1.log!AVG!1000000,2000000!5000000,
5000000!10

}
```

在上例中，“/var/lib/mrtg/192.168.1.253_1.log”参数传给 **check_local_mrtgtraf** 命令意思是插件的 MRTG 日志文件在这个文件里读写，“AVG”参数的意思是取带宽的统计平均值，“1000000,2000000”参数是指流入的告警门限(以字节为单位)，“5000000,5000000”是输出流量紧急状态门限(以字节为单位)，“10”是指如果 MRTG 日志如果超过 10 分钟没有数据返回一个紧急状态(应该每 5 分钟更新一次)。

保存该配置文件

4.9.11. 重启 Nagios

一旦给 **switch.cfg** 文件里加好新的主机与服务对象定义，就可以开始对路由器与交换机进行监控。为了开始监控，需要先 [验证配置文件](#)再 [重新启动Nagios](#)。

如果验证过程有任何错误信息，修改配置文件再继续。一定要保证配置验证过程中没有错误信息再启动 Nagios！

4.10. 监控网络打印机

4.10.1. 介绍

本文件描述了如何监控网络打印机。特别是有内置或外置 JetDirect 卡的 HP 惠普打印机设备，或是其他(象 Troy PocketPro 100S 或 Netgear PS101)支持 JetDirect 协议的打印机。

check_hpjd 插件(该命令是 Nagios 插件软件发行包的标准组成部分)可以用 SNMP 使能的方式来监控 JetDirect 兼容型打印机。该插件可以检查如下打印机状态：

- 卡纸
- 无纸
- 打印机离线
- 需要人工干预
- 墨盒墨粉低
- 内存不足
- 开外壳
- 输出托盘已满
- 和其他...

注意



如下指令假定你已经按照 [快速安装指南](#) 安装好 Nagios。可以参考安装好的样本配置文件 (`commands.cfg`、`templates.cfg` 等)。

4.10.2. 概览

监控网络打印机的状态很简单。有 JetDirect 功能的打印机一般提供 SNMP 功能，可以用 `check_hpjd` 插件来检测状态。

`check_hpjd` 插件只是当当前系统中安装有 `net-snmp` 和 `net-snmp-utils` 软件包时才会被编译和安装。要保证在 `/usr/local/nagios/libexec` 目录下有 `check_hpjd` 文件再继承，否则，要安装好 `net-snmp` 和 `net-snmp-utils` 软件包再重新编译安装 Nagios 插件包。

4.10.3. 步骤

监控打印机需要做如下几步：

- 做些事先准备工作；
- 创建一个用于监控打印机的主机与服务对象定义；
- 重启 Nagios 守护进程。

4.10.4. 已经做了什么？

为使这项工作更轻松，几个配置工作已经做好：

- `check_hpjd` 的命令定义已经加到了 `commands.cfg` 配置文件中，可以用 `check_hpjd` 插件来监控网络打印机；
- 一个网络打印机模板 (命名为 `generic-printer`) 已经在 `templates.cfg` 配置文件里创建好，可以更方便地加入一个新打印机设备的主机对象。

上面的监控配置文件可以在 `/usr/local/nagios/etc/objects/` 目录里找到。如果想做，可以修改里面的定义以更好地适用于你的情况。但是在此之前，推荐你要熟悉 Nagios 的配置之后再。起初，最好只是按下面的大概修改一下以实现网络打印机的监控。

4.10.5. 事先准备工作

在配置 Nagios 用于监控网络打印机之前，有些额外工作，记住这是要对第一台打印机设备进行监控。编辑 Nagios 的主配置文件。

```
vi /usr/local/nagios/etc/nagios.cfg
```

移除下面这行最前面的 (#) 号：

```
#cfg_file=/usr/local/nagios/etc/objects/printer.cfg
```

保存文件并退出编辑。

为何要这样？告诉 Nagios 查找 `/usr/local/nagios/etc/objects/printer.cfg` 文件以取得额外对象定义。该文件中将加入网络打印机设备的主机与服务对象定义。这个配置文件里已经包含有一个样本主机、主机组和服务定义。给第一台打印机设备做监控，可以简单地修改这个文件而不需重生成一个。

4.10.6. 配置 Nagios

需要创建几个 [对象定义](#) 以进行网络打印机的监控。

打开 `printer.cfg` 文件并编辑它。

```
vi /usr/local/nagios/etc/objects/printer.cfg
```

增加一个你要监控的网络打印机设备的 [主机对象定义](#)。如果这是第一台打印机设备，可以简单地修改 `printer.cfg` 文件里的样本主机定义。将合理的值赋在 `host_name`、`alias` 和 `address` 域里。

```
define host{
    use                generic-printer    ; Inherit default values from a template
    host_name          hplj2605dn        ; The name we're giving to this printer
    alias              HP LaserJet 2605dn ; A longer name associated with the
printer
    address            192.168.1.30      ; IP address of the printer
    hostgroups         allhosts ; Host groups this printer is associated with
}
```

现在可以给监控的打印机加些服务定义(在同一个配置文件里)。如果是第一台被监控的网络打印机，可以简单地修改 `printer.cfg` 里的服务配置。

注意



要用你要刚刚加上的被监控打印机主机名替换样例对象“`hplj2605dn`”里的 `host_name` 域值。

按如下方式加好对打印机状态检测的服务定义。服务用 `check_hpjd` 插件来检测打印机状态，默认情况下每 10 分钟检测一次。SNMP 共同体串是“`public`”。

```
define service{
    use                generic-service    ; Inherit values from a template
    host_name          hplj2605dn        ; The name of the host the service
is associated with
```

```

        service_description      Printer Status      ; The service description
        check_command            check_hpjd!-C public    ; The command used to monitor the
service
        normal_check_interval    10                    ; Check the service every 10 minutes under normal
conditions
        retry_check_interval     1                      ; Re-check the service every minute until its
final/hard state is determined
    }

```

加入一个默认每 10 分钟进行一次的 PING 检测服务。用于检测 RTA、丢包率和网络联接状态。

```

define service{
    use                generic-service
    host_name          hplj2605dn
    service_description PING
    check_command       check_ping!3000.0,80%!5000.0,100%
    normal_check_interval 10
    retry_check_interval 1
}

```

保存配置文件。

4.10.7. 重新启动 Nagios

一旦在 **printer.cfg** 文件里加好新的主机和服务对象定义就可以监控网络打印机。为了开始，应该先[验证配置文件](#)并[重新启动 Nagios](#)。

如果在验证配置过程中有任何错误信息，修改好配置文件再继续。保证验证过程完成且没有任何错误的情况下再重新启动 Nagios!

4.11. 监控 Netware 服务器

4.11.1. 介绍

本文档描述了如何对 Netware 服务器的“私有”服务和属性进行监控，象这些：

- 内存占用率
- 处理器利用率
- 缓冲区使用情况
- 活动的联接

- 磁盘卷使用率
- 等

由Netware服务器提供的公众服务(HTTP、FTP等)的监控可以按文档 [监控公众服务](#)来做。

4.11.2. 概览

TODO...

注意



我在找一个志愿者来写就HOWTO文档。我只能接触到一台旧的Netware 4.11 服务器，所以无法跟上形势需要。如果可以更新这个文档，请把它张贴到 [NagiosCommunity wiki](#)里。

4.11.3. 其他资源

Novell有一些有关Nagios如何做Netware监控的文档，在网站的 [Cool Solutions](#)栏目里，包括：

- [MRTGEXT: NLM module for MRTG and Nagios](#)
- [Nagios: Host and Service Monitoring Tool](#)
- [Nagios and NetWare: SNMP-based Monitoring](#)
- [Monitor DirXML/IDM Driver States with Nagios](#)
- [Check NDS Login ability with Nagios](#)
- [NDPS/iPrint Print Queue Monitoring by Nagios](#)
- [check_gwiaRL Plugin for Nagios 2.0](#)

4.12. 监控公众服务平台

4.12.1. 介绍

本文档介绍如何来监控一些公众化的服务、应用及协议。所谓的“公众”服务是指在网络里常见的服务—不管是本地网络还是因特网上都是这样。公众服务包括象—HTTP 服务(WEB)、POP3、IMAP/SMTP、FTP 和 SSH。其实在日常使用中还有更多的基础服务。这些服务与应用，包括所依托的协议，可以被 Nagios 直接监控而不需要额外的支持。

与之相对，私有服务如果没有某些中间件做代理 Nagios 是无法监控的。主机上的私有服务比如说 CPU 负荷、内存占用率、磁盘利用率、当前登录用户、进程信息等。这些私有服务或属性不能暴露给外部客户。这样就要在这些被监控的主机上安装一些中间件做代理来取得此类信息。更多有关在不同类型主机上的私有服务的信息可以查阅如下文档：

- [监控Windows主机](#)
- [监控Netware服务器](#)

- [监控Linux/Unix机器](#)

提示



偶尔发现有些私有服务和应用的信息可以通过SNMP来做监控。SNMP代理可以让远程监控系统提取一些有用信息。更多有关SNMP来监控服务信息的内容可以查阅一下 [监控交换机和路由器](#)。

注意



如下内容假定你已经按照 [快速安装指南](#) 安装好Nagios系统。如下的配置样例对象可以参考安装包里的 `commands.cfg` 和 `localhost.cfg` 两个配置文件。

4.12.2. 监控服务的插件

当需要对一些应用、服务或协议进行监控时，可能已经有一个用于监控的 [插件](#) 了。Nagios的官方插件包带有插件可以用于监控很多种服务与协议，在插件包的 `contrib/` 目录下有很多可用的插件。而且在 [NagiosExchange.org](#) 网站上有许多额外的其他人写的插件可以试试看！

如果不巧没有找到要监控所需的插件，可以自己写一个。插件写起来很容易，不要怕它，读一读 [插件的开发](#) 这篇文档吧。

下面将会领略一下你迟早会用到的一些基础服务，每个服务都可以在 Nagios 插件包里找到对应的检测插件。下面就开始了...

4.12.3. 创建一个主机对象定义

在监控一个服务之前需要先定义一个服务要绑定的 [主机](#) 对象。可以把主机对象定义放在 `cfg_file` 域所指向的任何一个对象配置文件里或是放在由 `cfg_dir` 域所指向的任何一个目录下的文件里。如果已经定义过一个主机对象了，可以跳过这一步。

在本例中，假定你想在一台主机上监控许多种服务，把这台主机命名为 `remotehost`。可以把主机对象定义放在它自己的配置文件里或加到一个已有的对象定义文件里。这里有个 `remotehost` 对象的定义象是这样：

```
define host{
    use                generic-host                ; Inherit default values from a template
    host_name          remotehost                    ; The name we're giving to this
host
    alias              Some Remote Host ; A longer name associated with the host
    address            192.168.1.50                ; IP address of the host
```

```
hostgroups          allhosts          ; Host groups this host is associated with
}
```

现在可以有了一个可被监控的主机对象了，之后就可以定义所要监控的服务。有了主机对象定义，服务对象定义可以加在任何一个对象配置文件的任何一个位置里。

4.12.4. 创建服务对象定义

在Nagios里每个要监控的服务都必须给出一个绑定在刚才定义出的主机上的一个 [服务对象](#)。可以把服务对象放在任何一个由 [cfg_file](#)域指向的对象配置文件里或是放在 [cfg_dir](#)域所指向的目录下。

下面会给出一些要监控的样例服务(HTTP、FTP 等)的对象定义。

4.12.5. 监控 HTTP

有时不管是你或是其他人需要监控一个 Web 服务器。**check_http** 插件就是做这件事的。插件可以透过 HTTP 协议并监控响应时间、错误代码、返回页面里的字符串、服务器证书和更多的东西。

在 **commands.cfg** 文件里包含有一个使用 **check_http** 插件的命令定义。象是这样的：

```
define command{
    name                check_http
    command_name        check_http
    command_line        $USER1$/check_http -I $HOSTADDRESS$ $ARG1$
}
```

对于监控在 **remotehost** 机器上的 HTTP 服务的简单的服务对象定义会象是这样的：

```
define service{
    use                generic-service          ; Inherit default values from a template
    host_name          remotehost
    service_description HTTP
    check_command       check_http
}
```

这个服务对象定义将会监控位于 **remotehost** 机器上的 HTTP 服务。如果 WEB 服务在 10 秒内没有响应将产生警报或是返回了一个 HTTP 的错误码(403、404 等)也会产生警报。这是最基本的监控要求，很简单，不是么？

提示



对于更高级的监控，可以在命令行下手工运行 **check_http** 插件，带着**--help** 参数，看看这个插件有什么选项。**--help** 语法对于本文档所涉及的插件都适用。

下面看看对 HTTP 服务更高级的监控。这个服务定义将会检测如下内容：在 URI (/download/index.php) 里是否包含有“latest-version.tar.gz”字符串。如果没有指定字符串，或是 URI 非法也或许是在 5 秒内没有响应的话，它将产生一个错误。

```
define service{
    use                generic-service        ; Inherit default values from a template
    host_name          remotehost
    service_description Product Download Link
    check_command       check_http!-u /download/index.php -t 5 -s "latest-version.tar.gz"
}
```

4.12.6. 监控 FTP

监控 FTP 服务可以用 **check_ftp** 插件。在 **commands.cfg** 文件里包含有一个使用 **check_ftp** 插件的命令样例，象是这样：

```
define command{
    command_name       check_ftp
    command_line       $USER1$/check_ftp -H $HOSTADDRESS$ $ARG1$
}
```

对 **remotehost** 主机上的 FTP 服务的简单监控的例子象是这样：

```
define service{
    use                generic-service        ; Inherit default values from a template
    host_name          remotehost
    service_description FTP
    check_command       check_ftp
}
```

这个服务对象定义将会在 FTP 服务器 10 秒内不响应时给出一个警报。

下面对 FTP 做些高级监控。下面这个服务将检测 FTP 服务器绑定在 **remotehost** 主机上的 1023 端口的 FTP 服务。如果在 5 秒内没有响应或是服务里没有回应字符串“Pure-FTPd [TLS]”时将会产生警报。

```
define service{
```

```

use                generic-service                ; Inherit default values from a template
host_name          remotehost
service_description Special FTP
check_command       check_ftp!-p 1023 -t 5 -e "Pure-FTPd [TLS]"
}

```

4.12.7. 监控 SSH

监控 SSH 服务可以用 **check_ssh** 插件。在 **commands.cfg** 文件里包含在使用 **check_ssh** 插件有样例命令，象是这样：

```

define command{
    command_name    check_ssh
    command_line     $USER1$/check_ssh $ARG1$ $HOSTADDRESS$
}

```

监控 **remotehost** 上的 SSH 服务的简单例子象是这样：

```

define service{
    use                generic-service                ; Inherit default values from a template
    host_name          remotehost
    service_description SSH
    check_command       check_ssh
}

```

这个服务对象定义将会在 SSH 服务 10 秒内不响应时给出一个警报。

下面对 SSH 做些高级监控。下面这个服务将检测 SSH 服务，如果 5 秒内不响应或是服务器版本串里没有能匹配字符串“OpenSSH_4.2”时产生一个警报。

```

define service{
    use                generic-service                ; Inherit default values from a template
    host_name          remotehost
    service_description SSH Version Check
    check_command       check_ssh!-t 5 -r "OpenSSH_4.2"
}

```

4.12.8. 监控 SMTP

用 **check_smtp** 插件来监控 EMail 服务。**commands.cfg** 文件里包含有使用 **check_smtp** 插件的命令，象是这样：

```
define command{
    command_name      check_smtp
    command_line      $USER1$/check_smtp -H $HOSTADDRESS$ $ARG1$
}
```

监控 **remotehost** 上的 SMTP 服务的简单例子象是这样：

```
define service{
    use                generic-service      ; Inherit default values from a template
    host_name          remotehost
    service_description SMTP
    check_command       check_smtp
}
```

这个服务对象定义将会在 SMTP 服务器 10 秒内不响应时给出一个警报。

下面是更高级的监控。下面这个服务将检测 SMTP 服务，如果服务在 5 秒内不响应或是返回串里没有包含字符串“mygreatmailserver.com”时将产生一个警报。

```
define service{
    use                generic-service      ; Inherit default values from a template
    host_name          remotehost
    service_description SMTP Response Check
    check_command       check_smtp!-t 5 -e "mygreatmailserver.com"
}
```

4.12.9. 监控 POP3

用 **check_pop** 插件来监控 EMail 的 POP3 服务。**commands.cfg** 文件里包含有使用 **check_pop** 插件的命令，象是这样：

```
define command{
    command_name      check_pop
    command_line      $USER1$/check_pop -H $HOSTADDRESS$ $ARG1$
}
```

监控 **remotehost** 上的 POP3 服务的简单例子象是这样：

```
define service{
    use                generic-service        ; Inherit default values from a template
    host_name          remotehost
    service_description POP3
    check_command      check_pop
}
```

这个服务对象定义将会在 POP3 服务 10 秒内不响应时给出一个警报。

下面是更高级监控。下面这个服务将检测 POP3 服务，当服务在 5 秒内不响应或是返回串里没有包含字符串“mygreatmailserver.com”时将产生一个警报。

```
define service{
    use                generic-service        ; Inherit default values from a template
    host_name          remotehost
    service_description POP3 Response Check
    check_command      check_pop!-t 5 -e "mygreatmailserver.com"
}
```

4.12.10. 监控 IMAP

用 **check_imap** 插件可以监控 EMail 的 IMAP4 服务。**commands.cfg** 文件里包含有使用 **check_imap** 插件的命令，象是这样：

```
define command{
    command_name      check_imap
    command_line      $USER1$/check_imap -H $HOSTADDRESS$ $ARG1$
}
```

监控 **remotehost** 上的 IMAP 服务的简单例子象是这样：

```
define service{
    use                generic-service        ; Inherit default values from a template
    host_name          remotehost
    service_description IMAP
    check_command      check_imap
}
```

```
}
```

这个服务对象定义将在 IMAP 服务 10 秒内不响应时给出一个警报。

下面是更高级监控。下面这个服务将检测 IMAP4 服务，当服务在 5 秒内不响应或是返回串里没有包含有“mygreatmailserver.com”时将产生一个警报。

```
define service{
    use                generic-service        ; Inherit default values from a template
    host_name          remotehost
    service_description IMAP4 Response Check
    check_command       check_imap!-t 5 -e "mygreatmailserver.com"
}
```

4.12.11. 重新启动 Nagios

一旦在配置文件里加入了一个新的主机或是服务对象，要开始对其进行监控，必须要 [验证配置文件](#) 并 [重新启动Nagios](#)。

如果验证过程里报出错误信息，要修正配置后再继续。要保证在验证过程里没有发现任何错误之后再启动或重启 Nagios！

第 5 章 准备配置 Nagios

第 5 章 准备配置 Nagios

5.1. 配置概览

5.1.1. 介绍

在你开始监控网络与系统之前要有同个不同配置文件需要创建和编辑。耐心点，配置 Nagios 可能是要花些时间特别是对于那些初次使用者。弄清其机理所有的将它们搞定绝对是值得的。 :-)

注意



样本配置文件在安装时放在了 `/usr/local/nagios/etc/` 目录下，如果你是按照前面给出的 [快速安装指南](#) 来做的话。

5.1.2. 主配置文件

主配置文件包括了一系列的设置，它们会影响 Nagios 守护进程。不仅是 Nagios 守护进程要使用主配置文件，CGIs 程序组模块也需要，因此，主配置文件是你开始学习配置其他文件的基础。

有关主配置文件的文档在 [这里](#)。

5.1.3. 资源配置文件

资源文件可以保存用户自定义的宏。资源文件的一个主要用处是用于保存一些敏感的配置信息如系统口令等不能让 CGIs 程序模块获取到的东西。

你可以在主配置文件中设置 [resource file](#) 指向一个或是多个资源文件。

5.1.4. 对象定义文件

对象定义文件用于定义主机、服务、主机组、服务组、联系人、联系人组、命令等等。这些将定义你需要监控什么并将如何监控它们。

你可以在主配置文件里设置 [cfg file](#) 加上 [cfg dir](#) 来指向一个或是多个对象定义文件。

有关对象定义和与其他间关系的文档是 [这里](#)。

5.1.5. CGI 配置文件

CGI 配置文件包含了一系列的设置，它们会影响 [CGIs](#) 程序模块。还有一些保存在主配置文件之中，因此 CGI 程序会知道你是如何配置的 Nagios 并且在哪里保存了对象定义。

有关 CGI 配置文件的文档在 [这里](#)。

5.2. 主配置文件选项

注意

当创建或编辑配置文件时，要遵守如下要求：

- 以符号 '#' 开头的行将视为注释不做处理；
- 变量必须是新起的一行 — 变量之前不能有空格符；
- 变量名是大小写敏感的；

提示



样例配置文件 (`/usr/local/nagios/etc/nagios.cfg`) 已经安装到位，如果你是按照 [快速安装指南](#) 来操作的话。

5.2.1. 配置文件的位置

主配置文件一般(实际是固定的)是 `nagios.cfg`，存放位置在 `/usr/local/nagios/etc/` 目录里(——如果是 rpm 包来安装，应该是在 `/etc/nagios/`)。

5.2.2. 配置文件里的变量

下面将对每个主配置文件里的选项进行说明...

表 5.1. 日志文件

格式:	<code>log_file=<file_name></code>
样例:	<code>log_file=/usr/local/nagios/var/nagios.log</code>

这个变量用于设定Nagios在何处创建其日志文件。它应该是你主配置文件里面的第一个变量，当Nagios找到你配置文件并发现配置里有错误时会向该文件中写入错误信息。如果你使能了[日志回滚](#)，Nagios将在每小时、每天、每周或每月对日志进行回滚。

表 5.2. 对象配置文件

格式:	<code>cfg_file=<file_name></code>
样例:	<code>cfg_file=/usr/local/nagios/etc/hosts.cfg</code> <code>cfg_file=/usr/local/nagios/etc/services.cfg</code> <code>cfg_file=/usr/local/nagios/etc/commands.cfg</code>

该变量用于指定一个包含有将用于Nagios监控对象的[对象配置文件](#)。对象配置文件中包括有主机、主机组、联系人、联系人组、服务、命令等等对象的定义。配置信息可以切分为多个文件并且用`cfg_file=`语句来指向每个待处理的配置文件。

表 5.3. 对象配置目录

格式:	<code>cfg_dir=<directory_name></code>
样例:	<code>cfg_dir=/usr/local/nagios/etc/commands</code> <code>cfg_dir=/usr/local/nagios/etc/services</code> <code>cfg_dir=/usr/local/nagios/etc/hosts</code>

该变量用于指定一个目录，目录里包含有将用于Nagios监控对象的[对象配置文件](#)。所有的在这个目录下的且以`.cfg`为扩展名的文件将被作为配置文件来处理。另外，Nagios将会递归该目录下的子目录并处理其子目录下的全部配置文件。你可以把配置放入不同的目录并且用`cfg_dir=`语句来指向每个待处理的目录。

表 5.4. 对象缓冲文件

格式:	<code>object_cache_file=<file_name></code>
样例:	<code>object_cache_file=/usr/local/nagios/var/objects.cache</code>

该变量用于指定一个用于缓冲 [对象定义](#) 复本的文件存放位置。对象缓冲将在每次Nagios的启动和重启时和使用CGI模块时被创建或重建。它试图加快在CGI里的配置缓冲并使得你在编辑 [对象配置文件](#) 时可以让正在运行的Nagios不影响CGI的显示输出。

表 5.5. 预缓冲对象文件

格式:	<code>precached_object_file=<file_name></code>
样例:	<code>precached_object_file=/usr/local/nagios/var/objects.precache</code>

该变量用于指定一个用于指定一个用于预处理、预缓冲 This directive is used to specify a file in which a pre-processed, pre-cached copy of [对象定义](#) 复本的文件存放位置。在大型或复杂Nagios安装模式下这个文件可用于显著地减少Nagios的启动时间。如何加快启动的更多信息可以查看 [这个](#) 内容。

表 5.6. 资源文件

格式:	<code>resource_file=<file_name></code>
样例:	<code>resource_file=/usr/local/nagios/etc/resource.cfg</code>

该变量用于指定一个可选的包含有\$USERn\$[宏](#)定义的可选资源文件。\$USERn\$宏在存放用户名、口令及通用的命令定义内容(如目录路径)时非常有用。CGIs模块将[不会](#)试图读取资源文件，所以你可以限定这权文件权限(600 或 660)来保护敏感信息。你可以在主配置文件里用resource_file语句来加入多个资源文件—Nagios将会处理它们。如何定义\$USERn\$宏参见样例resource.cfg文件，它放在Nagios发行包的sample-config/子目录下。

表 5.7. 临时文件

格式:	<code>temp_file=<file_name></code>
样例:	<code>temp_file=/usr/local/nagios/var/nagios.tmp</code>

该变量用于指定一个临时文件，Nagios 将在更新注释数据、状态数据等时周期性地创建它。该文件不再需要时会删除它。

表 5.8. 临时路径

格式:	<code>temp_path=<dir_name></code>
样例:	<code>temp_path=/tmp</code>

这个变量是一个目录，该目录是块飞地，在监控过程中用于创建临时文件。你应在该目录内运行 `tmpwatch` 或类似的工具程序以删除早于 24 小时的文件(这是个垃圾文件存放地)。

表 5.9. 状态文件

格式:	<code>status_file=<file_name></code>
样例:	<code>status_file=/usr/local/nagios/var/status.dat</code>

这个变量指向一个文件，文件被 Nagios 用于保存当前状态、注释和宕机信息。CGI 模块也会用这个文件以通过 Web 接口来显示当前被监控的状态，CGI 模块必须要有这个文件的读取权限以使工作正常。在 Nagios 停机或在重启动时将会删除并重建该文件。

表 5.10. 状态文件更新间隔

格式:	<code>status_update_interval=<seconds></code>
样例:	<code>status_update_interval=15</code>

这个变量设置了Nagios更新 [状态文件](#) 的速度(秒为单位)，最小更新间隔是 1 秒。

表 5.11. Nagios 用户

格式:	<code>nagios_user=<username/UID></code>
样例:	<code>nagios_user=nagios</code>

该变量指定了 Nagios 进程使用哪个用户运行。当程序启动完成并开始监控对象之前，Nagios 将切换自己的权限并使用该用户权限运行。你可以指定用户或是 UID 名。

表 5.12. Nagios 组

格式:	<code>nagios_group=<groupname/GID></code>
样例:	<code>nagios_group=nagios</code>

该变量用于指定 Nagios 使用哪个用户组运行。当程序启动完成并开始监控对象之前，Nagios 将切换自己的权限并以该用户组权限运行。你可以拽定用户组或 GID 名。

表 5.13. 通知选项

格式:	<code>enable_notifications=<0/1></code>
样例:	<code>enable_notifications=1</code>

该选项决定了Nagios在初始化启动或重启动时是否要送出 [通知](#)。如果这个选项不使能，Nagios将不会向任何主机或服务送出通知。注意，如果你打开了 [状态保持](#)选项，Nagios在其启动和重启时将忽略此设置并用这个选项的最近的一个设置(已经保存在 [状态保持文件](#))的值来工作，除非你取消了 [use_retained_program_state](#)选项。如果你想在使能状态保存选项(并且是 [use_retained_program_state](#)使能)的情况下更改这个选项，你必须要通过合适的 [外部命令](#)或是通过Web接口来修改它。选项的取值可以是：

- 0 = 关闭通知
- 1 = 打开通知(默认)

表 5.14. 服务检测执行选项

格式:	<code>execute_service_checks=<0/1></code>
样例:	<code>execute_service_checks=1</code>

这个选项指定了Nagios在初始的启动或重启时是否要执行服务检测。如果这个没有使能，Nagios将不会主动地执行任何服务的检测并且保持一系列的“静默”状态(它仍旧可以接收 [强制检测](#)除非你已经将 [accept_passive_service_checks](#)选项关闭)。这个选项经常用于备份被监控服务配置，被监控服务的配置备份在文档 [冗余安装](#)或设置成一个 [分布式](#)监控环境中有所描述。注意：如果你已经使能了 [状态保持](#)，Nagios在其启动或重启时将会忽略这个选项设置并使用和旧的设置值(旧值保存于 [状态保持文件](#))，除非你关闭了 [use_retained_program_state](#)选项。如果你想在状态保持使能(和 [use_retained_program_state](#)选项使能)的情况下修改这个选项，你只得用适当的 [外部命令](#)或是通过Web接口来修改它。选项可用的值有：

- 0 = 不执行服务检测
- 1 = 执行服务检测(默认)

表 5.15. 强制服务检测结果接受选项

格式:	<code>accept_passive_service_checks=<0/1></code>
-----	--

样例:	<code>accept_passive_service_checks=1</code>
-----	--

该选项决定了Nagios在其初始化启动或重启后是否要授受 [强制服务检测](#)，如果它关闭了，Nagios将不会接受任何强制服务检测结果。注意：如果你已经使能了 [状态保持](#)，Nagios在其启动或重启时将会忽略这个选项设置并使用和旧的设置值(旧值保存于 [状态保持文件](#))，除非你关闭了 [use_retained_program_state](#)选项。如果你想在状态保持使能(和 [use_retained_program_state](#)选项使能)的情况下修改这个选项，你只得用适当的 [外部命令](#)或是通过Web接口来修改它。选项可用的值有：

- 0 = 不接受强制服务检测结果
- 1 = 接受强制服务检测结果(默认)

表 5.16. 主机检测执行选项

格式:	<code>execute_host_checks=<0/1></code>
样例:	<code>execute_host_checks=1</code>

该选项将决定Nagios在初始地启动或重启时是否执行按需地和有规律规划检测。如果该选项不使能，那么Nagios将不会对任何主机进行检测，然而它仍旧可以接收 [强制主机检测](#)结果除非你已经将 [accept_passive_host_checks](#)选项关闭。该选项通常用于监控服务器的配置备份，详细信息请查看 [冗余安装](#)的配置，或是用于设置一个 [分布式](#)监控环境中。注意：如果你已经使能 [retain_state_information](#)状态保持选项，Nagios将在启动和重启时使用旧的选项值(保存于 [state_retention_file](#)状态保持文件中)而忽略此设置，除非你关闭了 [use_retained_program_state](#)选项。如果你想在保持选项使能(且 [use_retained_program_state](#)选项使能)的情况下修改这个选项，你只得用适当的 [外部命令](#)或是通过Web接口来修改它。选项可用的值有：

- 0 = 不执行主机检测
- 1 = 执行主机检测(默认)

表 5.17. 强制主机检测接受选项

格式:	<code>accept_passive_host_checks=<0/1></code>
样例:	<code>accept_passive_host_checks=1</code>

该选项决定了在Nagios初始启动或重启后是否要接受 [强制主机检测](#)结果。如果这个选项关闭，Nagios将不再接受任何强制主机检测结果。注意：如果你使能 [retain_state_information](#)状态保持选项，Nagios

将在启动或重新启动时使用旧的选项设置(保存于 [state retention file](#) 状态保持文件中)而忽略这个设置。

除非你已经关闭 [use retained program state](#) 选项。如果你想在保持选项使能(且

[use retained program state](#) 选项使能)的情况下修改这个选项, 你只得用适当的 [外部命令](#) 或是通过Web接口来修改它。选项可用的值有:

- 0 = 不接受强制主机检测结果
- 1 = 接受强制主机检测结果(默认)

表 5.18. 事件处理选项

格式:	<code>enable_event_handlers=<0/1></code>
样例:	<code>enable_event_handlers=1</code>

该选项决定了在Nagios初始启动或重启后是否要运行 [事件处理](#), 如果该选项关闭, Nagios将不做任何主机或服务的事件处理。注意: 如果你使能 [retain state information](#) 状态保持选项(保存于 [state retention file](#) 状态保持文件中)而忽略这个设置, 除非你已经关闭 [use retained program state](#) 选项。如果你想在保持选项使能(且 [use retained program state](#) 选项使能)的情况下修改这个选项, 你只得用适当的 [外部命令](#) 或是通过Web接口来修改它。选项可用的值有:

- 0 = 禁止事件处理
- 1 = 打开事件处理(默认)

表 5.19. 日志回滚方法

格式:	<code>log_rotation_method=<n/h/d/w/m></code>
样例:	<code>log_rotation_method=d</code>

该选项决定了你想让 Nagios 以何种方法回滚你的日志文件。可用的值有:

- n = None (不做日志回滚 — 这个是默认值)
- h = Hourly (每小时做一次日志回滚)
- d = Daily (每天午夜做日志回滚)
- w = Weekly (每周六午夜做日志回滚)
- m = Monthly (每月最后一天的午夜做日志回滚)

表 5.20. 日志打包路径

格式:	<code>log_archive_path=<path></code>
-----	--

样例:	<code>log_archive_path=/usr/local/nagios/var/archives/</code>
-----	---

该选项将指定一个用于存放回滚日志文件的保存路径。如果没有使用 [日志回滚](#) 功能时会忽略此设置。

表 5.21. 外部命令检查选项

格式:	<code>check_external_commands=<0/1></code>
样例:	<code>check_external_commands=1</code>

该选项决定了Nagios是否要检查存于 [命令文件](#) 里的将要执行的命令。这个选项在你计划通过Web接口来运行 [CGI命令](#) 时必须打开它。更多的关于外部命令的信息可以查阅 [这份文档](#)。

- 0 = 不做外部命令检测
- 1 = 检测外部命令 (默认值)

表 5.22. 外部命令检测间隔

格式:	<code>command_check_interval=<xxx>[s]</code>
样例:	<code>command_check_interval=1</code>

如果你指定了一个数字加一个“s”(如 30s)，那么外部检测命令的间隔是这个数值以**秒**为单位的时间间隔。如果没有用“s”，那么外部检测命令的间隔是以这个数值的“时间单位”的时间间隔，除非你把 [interval_length](#) 的值(下面有说明)从默认 60 给更改了，这个值的意思是 60s，即一分钟。

注意：将这个值设置为-1 可令Nagios尽可能频繁地对外命令进行检测。在进行其他任务之前，Nagios 每次都将会读入并处理保存于 [命令文件](#) 之中的全部命令以进行命令检查。更多的关于外部命令的信息可以查阅 [这份文档](#)。

表 5.23. 外部命令文件

格式:	<code>command_file=<file_name></code>
样例:	<code>command_file=/usr/local/nagios/var/rw/nagios.cmd</code>

这是一个Nagios用于外部命令检测处理的文件，[命令CGI程序模块](#)将命令写入该文件，外部命令文件实现成一个命名管道(先入先出)，在Nagios启动时创建它，并在关闭时删除它。如果在Nagios启动时该文件已经存在，那么Nagios会给出一个错误信息后中止。更多的关于外部命令的信息可以查阅 [这份文档](#)。

表 5.24. 外部命令缓冲队列数

格式:	<code>external_command_buffer_slots=<#></code>
样例:	<code>external_command_buffer_slots=512</code>

注意：这是个高级特性。该选项决定了Nagios将使用多少缓冲队列来缓存外部命令，外部命令是从一个工作线程从外部命令文件将命令读入的，但这些外部命令还没有被Nagios的主守护程序处理。缓冲中的每个位置可以处理一个外部命令，所以这个选项决定了有多少命令可以被缓冲处理。为了对一个有大量强制检测系统(比如 [分布式系统安装](#)) 进行安装时，你可能需要降低这个值。你要考虑使用MRTG工具来绘制外部命令缓冲的利用率图表，如何配置绘制图表可阅读 [这篇](#) 文档。

表 5.25. 互锁文件

格式:	<code>lock_file=<file_name></code>
样例:	<code>lock_file=/tmp/nagios.lock</code>

该选项指定了 Nagios 在以守护态运行(以 `-d` 命令行参数运行)时在哪个位置上创建互锁文件。该文件包含有运行 Nagios 的进程 id 值(PID)。

表 5.26. 状态保持选项

格式:	<code>retain_state_information=<0/1></code>
样例:	<code>retain_state_information=1</code>

该选项决定了Nagios是否要在程序的两次启动之间保存主机和服务的状态信息。如果你使能了这个选项，你应预先给出了 [state_retention_file](#) 变量的值，当选项使能时，Nagios将会在程序停止(或重启)时保存全部的主机和服务的状态信息并且会在启动时再次预读入保存的状态信息。

- 0 = 不保存状态保持信息
- 1 = 保留状态保持信息(默认)

表 5.27. 状态保持文件

格式:	<code>state_retention_file=<file_name></code>
样例:	<code>state_retention_file=/usr/local/nagios/var/retention.dat</code>

该文件用于在Nagios停止之前保存状态、停机时间和注释等信息。当Nagios重启时它会在开始监控工作之前使用保存于这个文件里的信息用于初始化主机与服务的状态。为使Nagios在程序的启动之间利用状态保持信息，你必须使能 [retain state information](#) 选项。

表 5.28. 自动状态保持的更新间隔

格式:	<code>retention_update_interval=<minutes></code>
样例:	<code>retention_update_interval=60</code>

该选项决定了Nagios需要以什么频度(分钟为单位)在正常操作时自动地保存状态保持信息。如果你把这个值设置为 0，Nagios将不会以规则的间隔保存状态保持数据，但是Nagios仍旧会在停机或重启之前做保存状态保持数据的工作。如果你关闭了状态保持功能(用 [retain state information](#) 选项设置)，这个选项值将无效。

表 5.29. 程序所用状态的使用选项

格式:	<code>use_retained_program_state=<0/1></code>
样例:	<code>use_retained_program_state=1</code>

这个设置将决定了Nagios是否要使用保存于状态保持文件之中的值以更新程序范围内的变量状态。有些程序范围内的变量的状态将在程序重启时被保存于状态保持文件之中，包括 [enable notifications](#)、[enable flap detection](#)、[enable event handlers](#)、[execute service checks](#)和 [accept passive service checks](#) 选项。如果你没有使用 [retain state information](#) 状态保持选项使能，这个选项将无效。

- 0 = 不使用程序变量的状态值
- 1 = 使用状态保持文件中的程序变量状态记录(默认)

表 5.30. 使用保持计划表信息选项

格式:	<code>use_retained_scheduling_info=<0/1></code>
样例:	<code>use_retained_scheduling_info=1</code>

该选项决定 Nagios 在重启时是否要使用主机和服务的保持计划表信息(下次检测时间)。如果增加了很多数量(或很大百分比)的主机和服务, 建议你在首次重启动 Nagios 时关闭选项, 因为这个选项将会使初始检测误入歧途。其他情况下你可以要使能这个选项。

- 0 = 不使用计划表信息
- 1 = 使用保存的计划表信息(默认)

表 5.31. 保持主机和服务属性掩码

格式:	<code>retained_host_attribute_mask=<number></code> <code>retained_service_attribute_mask=<number></code>
样例:	<code>retained_host_attribute_mask=0</code> <code>retained_service_attribute_mask=0</code>

警告: 这是个高级特性。你需要读一下源程序以看清楚它是如何起效果的。

该选项决定了哪个主机和服务的属性在程序重启时不会被保留。这些选项值是与指定的“MODATTR_”值进行按位与运算出的, MODATTR_在源程序的 include/common.h 里定义, 默认情况下, 全部主机和服务的属性都会被保持。

表 5.32. 保持进程属性掩码

格式:	<code>retained_process_host_attribute_mask=<number></code> <code>retained_process_service_attribute_mask=<number></code>
样例:	<code>retained_process_host_attribute_mask=0</code> <code>retained_process_service_attribute_mask=0</code>

警告: 这是个高级特性。你需要读一下源程序以看清楚它是如何起效果的。

该选项决定了哪个进程属性在程序重启时不会被保留。有两个属性掩码因为经常是主机和服务的进程属性可以分别被修改。例如, 主机检测在程序层面上被关闭, 而服务检测仍旧被打开。这些选项值是与指定的“MODATTR_”值进行按位与运算出的, MODATTR_在源程序的 include/common.h 里定义, 默认情况下, 全部主机和服务的属性都会被保持。

表 5.33. 保持联系人属性掩码

格式:	<code>retained_contact_host_attribute_mask=<number></code> <code>retained_contact_service_attribute_mask=<number></code>
-----	---

样例:	<pre>retained_contact_host_attribute_mask=0 retained_contact_service_attribute_mask=0</pre>
-----	---

警告：这是个高级特性。你需要读一下源程序以看清楚它是如何起效果的。

该选项决定了哪个联系人属性在程序重启时不会被保留。有两个属性掩码因为经常是主机和服务的联系人属性可以分别被修改。这些选项值是与指定的“MODATTR_”值进行按位与运算出的，MODATTR_在源程序的 include/common.h 里定义，默认情况下，全部主机和服务的属性都会被保持。

表 5.34. Syslog 日志选项

格式:	<code>use_syslog=<0/1></code>
样例:	<code>use_syslog=1</code>

该选项决定了是否将日志信息记录到本地的 Syslog 中。可用的值有：

- 0 = 不使用 Syslog 机制
- 1 = 使用 Syslog 机制

表 5.35. 通知记录日志选项

格式:	<code>log_notifications=<0/1></code>
样例:	<code>log_notifications=1</code>

该选项决定了是否将通知信息记录进行记录，如果有很多联系人或是有规律性的服务故障时，记录文件将会增长很快。使用这个选项来保存已发出的通知记录。

- 0 = 不记录通知
- 1 = 记录通知

表 5.36. 服务检测重试记录选项

格式:	<code>log_service_retries=<0/1></code>
样例:	<code>log_service_retries=1</code>

该选项决定了是否将服务检测重试进行记录。服务检测重试发生在服务检测结果返回一个异常状态信息之时，而且你已经配置Nagios在对故障出现时进行一次以上的服务检测重试。此时有服务状态被认为是处理“软”故障状态。当调试Nagios或对服务的 [事件处理](#)进行测试时记录下服务检测的重试是非常有用的。

- 0 = 不记录服务检测重试
- 1 = 记录服务检测重试

表 5.37. 主机检测重试记录选项

格式:	<code>log_host_retries=<0/1></code>
样例:	<code>log_host_retries=1</code>

该选项决定了是否将主机检测重试进行记录。当调试Nagios或对主机的 [事件处理](#)进行测试时记录下主机检测的重试是非常有用的。

- 0 = 不记录主机检测重试
- 1 = 记录主机检测重试

表 5.38. 事件处理记录选项

格式:	<code>log_event_handlers=<0/1></code>
样例:	<code>log_event_handlers=1</code>

该选项决定了是否将服务和主机的 [事件处理](#)进行记录。一旦发生服务或主机状态迁移时，可选的事件处理命令会被执行。当调试Nagios或首次尝试事件处理脚本时记录下事件处理是非常有用的。

- 0 = 不记录事件处理
- 1 = 记录事件处理

表 5.39. 初始状态记录选项

格式:	<code>log_initial_states=<0/1></code>
样例:	<code>log_initial_states=1</code>

该选项决定了 Nagios 是否要强行记录全部的主机和服务的初始状态，即便状态报告是 OK 也要记录。只是在第一次检测发现主机和服务有异常时才会记录下初始状态。如果想用应用程序扫描一段时间内的主机和服务状态以生成统计报告时，使能这个选项将有很有帮助。

- 0 = 不记录初始状态(默认)
- 1 = 记录初始状态

表 5.40. 外部命令记录选项

格式:	<code>log_external_commands=<0/1></code>
样例:	<code>log_external_commands=1</code>

该选项决定了Nagios是否要记录 [外部命令](#)，外部命令是从 [command file](#)外部命令文件中提取的。注意：这个选项并不控制是否要对 [强制服务检测](#)（一种外部命令类型）进行记录。为使能或关闭对强制服务检测的记录，使用 [log_passive_checks](#)强制检测记录选项。

- 0 = 不记录外部命令
- 1 = 记录外部命令(默认)

表 5.41. 强制检测记录选项

格式:	<code>log_passive_checks=<0/1></code>
样例:	<code>log_passive_checks=1</code>

该选项决定了Nagios是否要记录来自于 [command file](#)外部命令文件的 [强制主机和强制服务检测](#)命令。如果要设置一个 [分布式监控环境](#)或是计划在规整的基础上要对大量的强制检测的结果进行处理时，需要关闭这个选项以防止日志文件过份增长。

- 0 = 不记录强制检测
- 1 = 记录强制检测(默认)

表 5.42. 全局主机事件处理选项

格式:	<code>global_host_event_handler=<command></code>
样例:	<code>global_host_event_handler=log-host-event-to-db</code>

该选项指定了当每个主机状态迁移时需要执行的主机事件处理命令。全局事件处理命令将优于在每个主机定义的事件处理命令而立即执行。**命令**参数是在 [对象配置文件](#)里定义的命令的短名称。由 [event_handler_timeout](#)事件处理超时选项控制的这个命令可运行的最大次数。更多的有关事件处理的信息可以查阅 [这篇文档](#)。

表 5.43. 全局服务事件处理选项

格式:	<code>global_service_event_handler=<command></code>
样例:	<code>global_service_event_handler=log-service-event-to-db</code>

该选项指定了当每个服务状态迁移时需要执行的服务事件处理命令。全局事件处理命令将优于在每个服务定义的事件处理命令而立即执行。**命令**参数是在 [对象配置文件](#)里定义的命令的短名称。由 [event handler timeout](#)事件处理超时选项控制的这个命令可运行的最大次数。更多的有关事件处理的信息可以查阅 [这篇文档](#)。

表 5.44. 检测休止时间间隔

格式:	<code>sleep_time=<seconds></code>
样例:	<code>sleep_time=1</code>

它指定了 Nagios 在进行计划表的下一次服务或主机检测命令执行之前应该休止多少秒。注意 Nagios 只是在已经进行了服务故障的排队检测之后才会休止。

表 5.45. 服务检测迟滞间隔计数方法

格式:	<code>service_inter_check_delay_method=<n/d/s/x.xx></code>
样例:	<code>service_inter_check_delay_method=s</code>

该选项容许你控制服务检测将如何初始展开事件队列。 Using a "smart" delay calculation (the default) will cause Nagios to calculate an average check interval and spread initial checks of all services out over that interval, thereby helping to eliminate CPU load spikes. Using no delay is generally **not** recommended, as it will cause all service checks to be scheduled for execution at the same time. This means that you will generally have large CPU spikes when the services are all executed in parallel. More information on how to estimate how the inter-check delay affects service check scheduling can be found [here](#). Values are as follows:

- n = Don't use any delay - schedule all service checks to run immediately (i.e. at the same time!)
- d = Use a "dumb" delay of 1 second between service checks

- s = Use a "smart" delay calculation to spread service checks out evenly (default)
- x.xx = Use a user-supplied inter-check delay of x.xx seconds

表 5.46. 最大服务检测传播时间

格式:	<code>max_service_check_spread=<minutes></code>
样例:	<code>max_service_check_spread=30</code>

This option determines the maximum number of minutes from when Nagios starts that all services (that are scheduled to be regularly checked) are checked. This option will automatically adjust the [service inter check delay method](#) service inter-check delay method (if necessary) to ensure that the initial checks of all services occur within the timeframe you specify. In general, this option will not have an affect on service check scheduling if scheduling information is being retained using the [use retained scheduling info](#) use_retained_scheduling_info option. 默认值是 30 分钟。

表 5.47. 服务交错因子

格式:	<code>service_interleave_factor=<s x></code>
样例:	<code>service_interleave_factor=s</code>

This variable determines how service checks are interleaved. Interleaving allows for a more even distribution of service checks, reduced load on remote hosts, and faster overall detection of host problems. Setting this value to 1 is equivalent to not interleaving the service checks (this is how versions of Nagios previous to 0.0.5 worked). Set this value to s (smart) for automatic calculation of the interleave factor unless you have a specific reason to change it. The best way to understand how interleaving works is to watch the [status CGI](#) (detailed view) when Nagios is just starting. You should see that the service check results are spread out as they begin to appear. More information on how interleaving works can be found [here](#).

- x = A number greater than or equal to 1 that specifies the interleave factor to use. An interleave factor of 1 is equivalent to not interleaving the service checks.
- s = Use a "smart" interleave factor calculation (default)

表 5.48. 最大并发服务检测数

格式:	<code>max_concurrent_checks=<max_checks></code>
样例:	<code>max_concurrent_checks=20</code>

该选项可指定在任意给定时间里可被同时运行的服务检测命令的最大数量。如果指定这个值为 1，则说明不允许任何并行服务检测，如果指定为 0 (默认值) 则是对并行服务检测。你须按照可运行 Nagios 的机器上的机器资源情况修改这个值，因为它会直接影响系统最大负荷，它施加于系统 (处理器利用率、内存使用率等) 之上。更多的关于如何评估需要设置多少并行检测值的信息可以查阅 [这篇](#) 文档。

表 5.49. 检测结果的回收频度

格式:	<code>check_result_reaper_frequency=<frequency_in_seconds></code>
样例:	<code>check_result_reaper_frequency=5</code>

该选项控制检测结果的回收事件的处理频度 (**以秒为单位**)。从主机和服务的检测过程里“回收”事件处理结果将是对已经执行结束的检测。事件的构成在 Nagios 里是监控逻辑里的核心内容。

表 5.50. 最大检测结果回收时间段

格式:	<code>max_check_result_reaper_time=<seconds></code>
样例:	<code>max_check_result_reaper_time=30</code>

该选项决定主机和服务检测结果回收时对结果回收时间段的控制，这个值是个**以秒为单位**的最大时间跨度。从主机和服务的检测过程里“回收”事件处理结果将是对已经执行结束的检测。如果有许多结果要处理，回收事件过程将占用很长时间来完成它，这将延迟对新的主机和服务检测的执行。该选项可以限制从检测结果得到与回收处理之间的最大时间间隔以使 Nagios 可以完成对其他监控逻辑的转换处理。

表 5.51. 检测结果保存路径

格式:	<code>check_result_path=<path></code>
样例:	<code>check_result_path=/var/spool/nagios/checkresults</code>

该选项决定了 Nagios 将在处理检测结果之前使用哪个目录来保存主机和服务检测结果。这个目录不能保存其他文件，因为 Nagios 会周期性地清理这个目录下的旧文件 (更多信息见 [max check result file age](#) 选项)。

注意：确保只有一个 Nagios 的实例在操作检测结果保存路径。如果有多个 Nagios 的实例来操作相同的目录，将会因为错误的 Nagios 实例不正确地处理导致有错误结果！

表 5.52. 检测结果文件的最大生存时间

格式:	<code>max_check_result_file_age=<seconds></code>
样例:	<code>max_check_result_file_age=3600</code>

该选项决定用最大多少秒来限定那些在 [check_result_path](#) 设置所指向目录里的检测结果文件是合法的。如果检测结果文件超出了这个门限，Nagios 将会把过旧的文件删除而且不会处理内含的检测结果。若设置该选项为 0，Nagios 将处理全部的检测结果文件—即便这些文件比你的硬件还老旧。

表 5.53. 主机检测迟滞间隔计数方式

格式:	<code>host_inter_check_delay_method=<n/d/s/x.xx></code>
样例:	<code>host_inter_check_delay_method=s</code>

This option allows you to control how host checks **that are scheduled to be checked on a regular basis** are initially “spread out” in the event queue. Using a “smart” delay calculation (the default) will cause Nagios to calculate an average check interval and spread initial checks of all hosts out over that interval, thereby helping to eliminate CPU load spikes. Using no delay is generally **not** recommended. Using no delay will cause all host checks to be scheduled for execution at the same time. More information on how to estimate how the inter-check delay affects host check scheduling can be found [here](#). Values are as follows:

- n = Don’t use any delay – schedule all host checks to run immediately (i.e. at the same time!)
- d = Use a “dumb” delay of 1 second between host checks
- s = Use a “smart” delay calculation to spread host checks out evenly (default)
- x.xx = Use a user-supplied inter-check delay of x.xx seconds

表 5.54. 最大主机检测传播时间

格式:	<code>max_host_check_spread=<minutes></code>
样例:	<code>max_host_check_spread=30</code>

This option determines the maximum number of minutes from when Nagios starts that all hosts (that are scheduled to be regularly checked) are checked. This option will automatically adjust the [host inter check delay method](#) host inter-check delay method (if necessary) to ensure that the initial checks of all hosts occur within the timeframe you specify. In general, this option will not have an affect on host check scheduling if scheduling information is being retained using the [use retained scheduling info](#) use_retained_scheduling_info option. Default value is 30 (minutes).

表 5.55. 计数间隔长度

格式:	<code>interval_length=<seconds></code>
样例:	<code>interval_length=60</code>

该选项指定了“单位间隔”是多少秒数，单位间隔用于计数计划队列处理、再次通知等。单位间隔在对象配置文件被用于决定以何频度运行服务检测、以何频度与联系人再通知等。

重要：默认值是 60，这说明在对象配置文件里设定的“单位间隔”是 60 秒(1 分钟)。我没测试过其他值，所以如果要用其他值要自担风险！

表 5.56. 自动计划检测选项

格式:	<code>auto_reschedule_checks=<0/1></code>
样例:	<code>auto_reschedule_checks=1</code>

该选项决定了 Nagios 是否要试图自动地进行计划的自主检测主机与服务以使在之后的时间里检测更为“平滑”。这可以使得监控主机保持一个均衡的负载，也使得在持续检测之间的保持相对一致，其代价是要更刚性地按计划执行检测工作。

WARNING: THIS IS AN EXPERIMENTAL FEATURE AND MAY BE REMOVED IN FUTURE VERSIONS. ENABLING THIS OPTION CAN DEGRADE PERFORMANCE – RATHER THAN INCREASE IT – IF USED IMPROPERLY!

表 5.57. Auto-Rescheduling Interval

格式:	<code>auto_rescheduling_interval=<seconds></code>
样例:	<code>auto_rescheduling_interval=30</code>

- 1 = Use aggressive host checking

表 5.60. 传递强制主机检测结果选项

格式:	<code>translate_passive_host_checks=<0/1></code>
样例:	<code>translate_passive_host_checks=1</code>

This option determines whether or not Nagios will DOWN/UNREACHABLE passive host check results to their "correct" state from the viewpoint of the local Nagios instance. This can be very useful in distributed and failover monitoring installations. More information on passive check state translation can be found [here](#).

- 0 = Disable check translation (default)
- 1 = Enable check translation

表 5.61. Passive Host Checks Are SOFT Option

格式:	<code>passive_host_checks_are_soft=<0/1></code>
样例:	<code>passive_host_checks_are_soft=1</code>

This option determines whether or not Nagios will treat [passive host checks](#) as HARD states or SOFT states. By default, a passive host check result will put a host into a [HARD state type](#). You can change this behavior by enabling this option.

- 0 = Passive host checks are HARD (default)
- 1 = Passive host checks are SOFT

表 5.62. Predictive Host Dependency Checks Option

格式:	<code>enable_predictive_host_dependency_checks=<0/1></code>
样例:	<code>enable_predictive_host_dependency_checks=1</code>

This option determines whether or not Nagios will execute predictive checks of hosts that are being depended upon (as defined in [host dependencies](#)) for a particular host when it changes state.

Predictive checks help ensure that the dependency logic is as accurate as possible. More information on how predictive checks work can be found [here](#).

- 0 = Disable predictive checks
- 1 = Enable predictive checks (default)

表 5.63. Predictive Service Dependency Checks Option

格式:	<code>enable_predictive_service_dependency_checks=<0/1></code>
样例:	<code>enable_predictive_service_dependency_checks=1</code>

This option determines whether or not Nagios will execute predictive checks of services that are being depended upon (as defined in [service dependencies](#)) for a particular service when it changes state.

Predictive checks help ensure that the dependency logic is as accurate as possible. More information on how predictive checks work can be found [here](#).

- 0 = Disable predictive checks
- 1 = Enable predictive checks (default)

表 5.64. Cached Host Check Horizon

格式:	<code>cached_host_check_horizon=<seconds></code>
样例:	<code>cached_host_check_horizon=15</code>

This option determines the maximum amount of time (in seconds) that the state of a previous host check is considered current. Cached host states (from host checks that were performed more recently than the time specified by this value) can improve host check performance immensely. Too high of a value for this option may result in (temporarily) inaccurate host states, while a low value may result in a performance hit for host checks. Use a value of 0 if you want to disable host check caching. More information on cached checks can be found [here](#).

表 5.65. Cached Service Check Horizon

格式:	<code>cached_service_check_horizon=<seconds></code>
样例:	<code>cached_service_check_horizon=15</code>

This option determines the maximum amount of time (in seconds) that the state of a previous service check is considered current. Cached service states (from service checks that were performed more recently than the time specified by this value) can improve service check performance when a lot of [service dependencies](#) are used. Too high of a value for this option may result in inaccuracies in the service dependency logic. Use a value of 0 if you want to disable service check caching. More information on cached checks can be found [here](#).

表 5.66. Large Installation Tweaks Option

格式:	<code>use_large_installation_tweaks=<0/1></code>
样例:	<code>use_large_installation_tweaks=0</code>

This option determines whether or not the Nagios daemon will take several shortcuts to improve performance. These shortcuts result in the loss of a few features, but larger installations will likely see a lot of benefit from doing so. More information on what optimizations are taken when you enable this option can be found [here](#).

- 0 = Don't use tweaks (default)
- 1 = Use tweaks

表 5.67. 子进程内存选项

格式:	<code>free_child_process_memory=<0/1></code>
样例:	<code>free_child_process_memory=0</code>

This option determines whether or not Nagios will free memory in child processes when they are fork()ed off from the main process. By default, Nagios frees memory. However, if the [use large installation tweaks](#) option is enabled, it will not. By defining this option in your configuration file, you are able to override things to get the behavior you want.

- 0 = Don't free memory
- 1 = Free memory

表 5.68. 子进程二次派生选项

格式:	<code>child_processes_fork_twice=<0/1></code>
-----	---

样例:	<code>child_processes_fork_twice=0</code>
-----	---

This option determines whether or not Nagios will fork() child processes twice when it executes host and service checks. By default, Nagios fork()s twice. However, if the [use large installation tweaks](#) option is enabled, it will only fork() once. By defining this option in your configuration file, you are able to override things to get the behavior you want.

- 0 = Fork() just once
- 1 = Fork() twice

表 5.69. 环境变量中标准宏可用性选项

格式:	<code>enable_environment_macros=<0/1></code>
样例:	<code>enable_environment_macros=0</code>

This option determines whether or not the Nagios daemon will make all standard [macros](#) available as environment variables to your check, notification, event handler, etc. commands. In large Nagios installations this can be problematic because it takes additional memory and (more importantly) CPU to compute the values of all macros and make them available to the environment.

- 0 = Don't make macros available as environment variables
- 1 = Make macros available as environment variables (default)

表 5.70. Flap Detection Option

格式:	<code>enable_flap_detection=<0/1></code>
样例:	<code>enable_flap_detection=0</code>

This option determines whether or not Nagios will try and detect hosts and services that are "flapping". Flapping occurs when a host or service changes between states too frequently, resulting in a barrage of notifications being sent out. When Nagios detects that a host or service is flapping, it will temporarily suppress notifications for that host/service until it stops flapping. Flap detection is very experimental at this point, so use this feature with caution! More information on how flap detection and handling works can be found [here](#). 注意：如果你使能

[retain state information](#)状态保持选项(保存于 [state retention file](#)状态保持文件中)而忽略这个设置, 除非你已经关闭 [use retained program state](#)选项。如果你想在保持选项使能(且 [use retained program state](#)选项使能)的情况下修改这个选项, 你只得用适当的 [外部命令](#)或是通过Web接口来修改它。选项可用的值有:

- 0 = Don't enable flap detection (default)
- 1 = Enable flap detection

表 5.71. Low Service Flap Threshold

格式:	<code>low_service_flap_threshold=<percent></code>
样例:	<code>low_service_flap_threshold=25.0</code>

This option is used to set the low threshold for detection of service flapping. For more information on how flap detection and handling works (and how this option affects things) read [this](#).

表 5.72. High Service Flap Threshold

格式:	<code>high_service_flap_threshold=<percent></code>
样例:	<code>high_service_flap_threshold=50.0</code>

This option is used to set the low threshold for detection of service flapping. For more information on how flap detection and handling works (and how this option affects things) read [this](#).

表 5.73. Low Host Flap Threshold

格式:	<code>low_host_flap_threshold=<percent></code>
样例:	<code>low_host_flap_threshold=25.0</code>

This option is used to set the low threshold for detection of host flapping. For more information on how flap detection and handling works (and how this option affects things) read [this](#).

表 5.74. High Host Flap Threshold

格式:	<code>high_host_flap_threshold=<percent></code>
样例:	<code>high_host_flap_threshold=50.0</code>

This option is used to set the low threshold for detection of host flapping. For more information on how flap detection and handling works (and how this option affects things) read [this](#).

表 5.75. Soft State Dependencies Option

格式:	<code>soft_state_dependencies=<0/1></code>
样例:	<code>soft_state_dependencies=0</code>

This option determines whether or not Nagios will use soft state information when checking [host and service dependencies](#). Normally Nagios will only use the latest hard host or service state when checking dependencies. If you want it to use the latest state (regardless of whether its a soft or hard [state type](#)), enable this option.

- 0 = Don't use soft state dependencies (default)
- 1 = Use soft state dependencies

表 5.76. 服务检测超时

格式:	<code>service_check_timeout=<seconds></code>
样例:	<code>service_check_timeout=60</code>

This is the maximum number of seconds that Nagios will allow service checks to run. If checks exceed this limit, they are killed and a 紧急 state is returned. A timeout error will also be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off plugins which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each service check normally finishes executing within this time limit. If a service check runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

表 5.77. 主机检测超时

格式:	<code>host_check_timeout=<seconds></code>
样例:	<code>host_check_timeout=60</code>

This is the maximum number of seconds that Nagios will allow host checks to run. If checks exceed this limit, they are killed and a 紧急 state is returned and the host will be assumed to be DOWN. A timeout error will also be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off plugins which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each host check normally finishes executing within this time limit. If a host check runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

表 5.78. 事件处理超时

格式:	<code>event_handler_timeout=<seconds></code>
样例:	<code>event_handler_timeout=60</code>

This is the maximum number of seconds that Nagios will allow [event handlers](#) to be run. If an event handler exceeds this time limit it will be killed and a warning will be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off commands which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each event handler command normally finishes executing within this time limit. If an event handler runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

表 5.79. 通知超时

格式:	<code>notification_timeout=<seconds></code>
样例:	<code>notification_timeout=60</code>

This is the maximum number of seconds that Nagios will allow notification commands to be run. If a notification command exceeds this time limit it will be killed and a warning will be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off commands which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each notification command finishes executing within this time limit. If a notification command runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

表 5.80. Obsessive Compulsive Service Processor Timeout

格式:	<code>ocsp_timeout=<seconds></code>
样例:	<code>ocsp_timeout=5</code>

This is the maximum number of seconds that Nagios will allow an [ocsp_command](#)obsessive compulsive service processor command to be run. If a command exceeds this time limit it will be killed and a warning will be logged.

表 5.81. Obsessive Compulsive Host Processor Timeout

格式:	<code>ochp_timeout=<seconds></code>
样例:	<code>ochp_timeout=5</code>

This is the maximum number of seconds that Nagios will allow an [ochp_command](#)obsessive compulsive host processor command to be run. If a command exceeds this time limit it will be killed and a warning will be logged.

表 5.82. 性能数据处理命令超时

格式:	<code>perfdata_timeout=<seconds></code>
样例:	<code>perfdata_timeout=5</code>

This is the maximum number of seconds that Nagios will allow a [host_perfdata_command](#)host performance data processor command or [service_perfdata_command](#)service performance data

processor command to be run. If a command exceeds this time limit it will be killed and a warning will be logged.

表 5.83. Obsess Over Services Option

格式:	obsess_over_services=<0/1>
样例:	obsess_over_services=1

This value determines whether or not Nagios will “obsess” over service checks results and run the [ocsp_command](#)obsessive compulsive service processor command you define. I know – funny name, but it was all I could think of. This option is useful for performing [distributed monitoring](#). If you’re not doing distributed monitoring, don’t enable this option.

- 0 = Don’t obsess over services (default)
- 1 = Obsess over services

表 5.84. Obsessive Compulsive Service Processor Command

格式:	ocsp_command=<command>
样例:	ocsp_command=obsessive_service_handler

This option allows you to specify a command to be run after **every** service check, which can be useful in [distributed monitoring](#). This command is executed after any [event handler](#) or [notification](#) commands. The **command** argument is the short name of a [command definition](#) that you define in your 对象配置文件. The maximum amount of time that this command can run is controlled by the [ocsp_timeout](#)ocsp_timeout option. More information on distributed monitoring can be found [here](#). This command is only executed if the [obsess over services](#)obsess_over_services option is enabled globally and if the **obsess_over_service** directive in the [service definition](#) is enabled.

表 5.85. Obsess Over Hosts Option

格式:	obsess_over_hosts=<0/1>
样例:	obsess_over_hosts=1

This value determines whether or not Nagios will “obsess” over host checks results and run the [ochp_command](#)obsessive compulsive host processor command you define. I know – funny name, but it was all I could think of. This option is useful for performing [distributed monitoring](#). If you’re not doing distributed monitoring, don’t enable this option.

- 0 = Don’t obsess over hosts (default)
- 1 = Obsess over hosts

表 5.86. Obsessive Compulsive Host Processor Command

格式:	<code>ochp_command=<command></code>
样例:	<code>ochp_command=obsessive_host_handler</code>

This option allows you to specify a command to be run after **every** host check, which can be useful in [distributed monitoring](#). This command is executed after any [event handler](#) or [notification](#) commands. The **command** argument is the short name of a [command definition](#) that you define in your 对象配置文件. The maximum amount of time that this command can run is controlled by the [ochp_timeout](#)ochp_timeout option. More information on distributed monitoring can be found [here](#). This command is only executed if the [obsess_over_hosts](#)obsess_over_hosts option is enabled globally and if the **obsess_over_host** directive in the [host definition](#) is enabled.

表 5.87. 性能数据处理选项

格式:	<code>process_performance_data=<0/1></code>
样例:	<code>process_performance_data=1</code>

该选项决定Nagios是否要处理主机和服务检测 [性能数据](#)。

- 0 = Don’t process performance data (default)
- 1 = Process performance data

表 5.88. 主机性能数据处理命令

格式:	<code>host_perfdata_command=<command></code>
样例:	<code>host_perfdata_command=process-host-perfdata</code>

This option allows you to specify a command to be run after **every** host check to process host [performance data](#) that may be returned from the check. The **command** argument is the short name of a [command definition](#) that you define in your 对象配置文件. This command is only executed if the [process performance data](#) `process_performance_data` option is enabled globally and if the **process_perf_data** directive in the [host definition](#) is enabled.

表 5.89. 服务性能数据处理命令

格式:	<code>service_perfdata_command=<command></code>
样例:	<code>service_perfdata_command=process-service-perfdata</code>

This option allows you to specify a command to be run after **every** service check to process service [performance data](#) that may be returned from the check. The **command** argument is the short name of a [command definition](#) that you define in your 对象配置文件. This command is only executed if the [process performance data](#) `process_performance_data` option is enabled globally and if the **process_perf_data** directive in the [service definition](#) is enabled.

表 5.90. 主机性能数据文件

格式:	<code>host_perfdata_file=<file_name></code>
样例:	<code>host_perfdata_file=/usr/local/nagios/var/host-perfdata.dat</code>

This option allows you to specify a file to which host [performance data](#) will be written after every host check. Data will be written to the performance file as specified by the [host perfdata file template](#) `host_perfdata_file_template` option. Performance data is only written to this file if the [process performance data](#) `process_performance_data` option is enabled globally and if the **process_perf_data** directive in the [host definition](#) is enabled.

表 5.91. 服务性能数据文件

格式:	<code>service_perfdata_file=<file_name></code>
样例:	<code>service_perfdata_file=/usr/local/nagios/var/service-perfdata.dat</code>

This option allows you to specify a file to which service [performance data](#) will be written after every service check. Data will be written to the performance file as specified by the [service_perfdata_file_template](#) option. Performance data is only written to this file if the [process_performance_data](#) process_performance_data option is enabled globally and if the `process_perf_data` directive in the [service definition](#) is enabled.

表 5.92. 主机性能数据文件模板

格式:	host_perfdata_file_template=<template>
样例:	host_perfdata_file_template=[HOSTPERFDATA]\t\$TIMET\$\t\$HOSTNAME\$\t\$HOSTEXECUTIONTIME\$\t\$HOSTOUTPUT\$\t\$HOSTPERFDATA\$

This option determines what (and how) data is written to the [host_perfdata_file](#) host performance data file. The template may contain [macros](#), special characters (\t for tab, \r for carriage return, \n for newline) and plain text. A newline is automatically added after each write to the performance data file.

表 5.93. 服务性能数据文件模板

格式:	service_perfdata_file_template=<template>
样例:	service_perfdata_file_template=[SERVICEPERFDATA]\t\$TIMET\$\t\$HOSTNAME\$\t\$SERVICEDESC\$\t\$SERVICEEXECUTIONTIME\$\t\$SERVICELATENCY\$\t\$SERVICEOUTPUT\$\t\$SERVICEPERFDATA\$

This option determines what (and how) data is written to the [service performance data file](#). The template may contain [macros](#), special characters (\t for tab, \r for carriage return, \n for newline) and plain text. A newline is automatically added after each write to the performance data file.

表 5.94. 主机性能数据文件打开方式

格式:	host_perfdata_file_mode=<mode>
-----	--------------------------------

样例:	<code>host_perfdata_file_mode=a</code>
-----	--

This option determines how the [host_perfdata_file](#) host performance data file is opened. Unless the file is a named pipe you'll probably want to use the default mode of append.

- a = Open file in append mode (default)
- w = Open file in write mode
- p = Open in non-blocking read/write mode (useful when writing to pipes)

表 5.95. 性能数据文件打开方式

格式:	<code>service_perfdata_file_mode=<mode></code>
样例:	<code>service_perfdata_file_mode=a</code>

This option determines how the [service performance data file](#) is opened. Unless the file is a named pipe you'll probably want to use the default mode of append.

- a = Open file in append mode (default)
- w = Open file in write mode
- p = Open in non-blocking read/write mode (useful when writing to pipes)

表 5.96. 主机性能数据文件处理间隔

格式:	<code>host_perfdata_file_processing_interval=<seconds></code>
样例:	<code>host_perfdata_file_processing_interval=0</code>

This option allows you to specify the interval (in seconds) at which the [host_perfdata_file](#) host performance data file is processed using the [host_perfdata_file_processing_command](#) host performance data file processing command. A value of 0 indicates that the performance data file should not be processed at regular intervals.

表 5.97. 服务性能数据文件处理间隔

格式:	<code>service_perfdata_file_processing_interval=<seconds></code>
样例:	<code>service_perfdata_file_processing_interval=0</code>

This option allows you to specify the interval (in seconds) at which the [service_perfdata_file](#) service performance data file is processed using the [service_perfdata_file_processing_command](#) service performance data file processing command. A value of 0 indicates that the performance data file should not be processed at regular intervals.

表 5.98. 主机性能数据文件处理命令

格式:	host_perfdata_file_processing_command=<command>
样例:	host_perfdata_file_processing_command=process-host-perfdata-file

This option allows you to specify the command that should be executed to process the [host_perfdata_file](#) host performance data file. The **command** argument is the short name of a [command definition](#) that you define in your 对象配置文件. The interval at which this command is executed is determined by the [host_perfdata_file_processing_interval](#) host_perfdata_file_processing_interval directive.

表 5.99. 服务性能数据文件处理命令

格式:	service_perfdata_file_processing_command=<command>
样例:	service_perfdata_file_processing_command=process-service-perfdata-file

This option allows you to specify the command that should be executed to process the [service_perfdata_file](#) service performance data file. The **command** argument is the short name of a [command definition](#) that you define in your 对象配置文件. The interval at which this command is executed is determined by the [service_perfdata_file_processing_interval](#) service_perfdata_file_processing_interval directive.

表 5.100. 孤立服务检测选项

格式:	check_for_orphaned_services=<0/1>
样例:	check_for_orphaned_services=1

This option allows you to enable or disable checks for orphaned service checks. Orphaned service checks are checks which have been executed and have been removed from the event queue, but have not had any results reported in a long time. Since no results have come back in for the service, it is not rescheduled in the event queue. This can cause service checks to stop being executed. Normally it is very rare for this to happen – it might happen if an external user or process killed off the process that was being used to execute a service check. If this option is enabled and Nagios finds that results for a particular service check have not come back, it will log an error message and reschedule the service check. If you start seeing service checks that never seem to get rescheduled, enable this option and see if you notice any log messages about orphaned services.

- 0 = Don't check for orphaned service checks
- 1 = Check for orphaned service checks (default)

表 5.101. 孤立主机检测选项

格式:	<code>check_for_orphaned_hosts=<0/1></code>
样例:	<code>check_for_orphaned_hosts=1</code>

This option allows you to enable or disable checks for orphaned host checks. Orphaned host checks are checks which have been executed and have been removed from the event queue, but have not had any results reported in a long time. Since no results have come back in for the host, it is not rescheduled in the event queue. This can cause host checks to stop being executed. Normally it is very rare for this to happen – it might happen if an external user or process killed off the process that was being used to execute a host check. If this option is enabled and Nagios finds that results for a particular host check have not come back, it will log an error message and reschedule the host check. If you start seeing host checks that never seem to get rescheduled, enable this option and see if you notice any log messages about orphaned hosts.

- 0 = Don't check for orphaned host checks
- 1 = Check for orphaned host checks (default)

表 5.102. 服务更新检测选项

格式:	<code>check_service_freshness=<0/1></code>
样例:	<code>check_service_freshness=0</code>

This option determines whether or not Nagios will periodically check the “freshness” of service checks. Enabling this option is useful for helping to ensure that [passive service checks](#) are received in a timely manner. More information on freshness checking can be found [here](#).

- 0 = Don't check service freshness
- 1 = Check service freshness (default)

表 5.103. 服务更新检测间隔

格式:	<code>service_freshness_check_interval=<seconds></code>
样例:	<code>service_freshness_check_interval=60</code>

This setting determines how often (in seconds) Nagios will periodically check the “freshness” of service check results. If you have disabled service freshness checking (with the [check_service_freshness](#) option), this option has no effect. More information on freshness checking can be found [here](#).

表 5.104. 主机更新检测选项

格式:	<code>check_host_freshness=<0/1></code>
样例:	<code>check_host_freshness=0</code>

This option determines whether or not Nagios will periodically check the “freshness” of host checks. Enabling this option is useful for helping to ensure that [passive host checks](#) are received in a timely manner. More information on freshness checking can be found [here](#).

- 0 = Don't check host freshness
- 1 = Check host freshness (default)

表 5.105. 主机更新检测间隔

格式:	<code>host_freshness_check_interval=<seconds></code>
-----	--

样例:	<code>host_freshness_check_interval=60</code>
-----	---

This setting determines how often (in seconds) Nagios will periodically check the “freshness” of host check results. If you have disabled host freshness checking (with the [check_host_freshness](#) option), this option has no effect. More information on freshness checking can be found [here](#).

表 5.106. Additional Freshness Threshold Latency Option

格式:	<code>additional_freshness_latency=<#></code>
样例:	<code>additional_freshness_latency=15</code>

This option determines the number of seconds Nagios will add to any host or services freshness threshold it automatically calculates (e.g. those not specified explicitly by the user). More information on freshness checking can be found [here](#).

表 5.107. Embedded Perl Interpreter Option

格式:	<code>enable_embedded_perl=<0/1></code>
样例:	<code>enable_embedded_perl=1</code>

This setting determines whether or not the embedded Perl interpreter is enabled on a program-wide basis. Nagios must be compiled with support for embedded Perl for this option to have an effect. More information on the embedded Perl interpreter can be found [here](#).

表 5.108. Embedded Perl Implicit Use Option

格式:	<code>use_embedded_perl_implicitly=<0/1></code>
样例:	<code>use_embedded_perl_implicitly=1</code>

This setting determines whether or not the embedded Perl interpreter should be used for Perl plugins/scripts that do not explicitly enable/disable it. Nagios must be compiled with support for embedded Perl for this option to have an effect. More information on the embedded Perl interpreter and the effect of this setting can be found [here](#).

表 5.109. Date Format

格式:	<code>date_format=<option></code>
样例:	<code>date_format=us</code>

This option allows you to specify what kind of date/time format Nagios should use in the web interface and date/time [macros](#). Possible options (along with example output) include:

表 5.110.

选项	输出格式	输出样例
us	MM/DD/YYYY HH:MM:SS	06/30/2002 03:15:00
euro	DD/MM/YYYY HH:MM:SS	30/06/2002 03:15:00
iso8601	YYYY-MM-DD HH:MM:SS	2002-06-30 03:15:00
strict-iso8601	YYYY-MM-DDTHH:MM:SS	2002-06-30T03:15:00

表 5.111. 时区选项

格式:	<code>use_timezone=<tz></code>
样例:	<code>use_timezone=US/Mountain</code>

This option allows you to override the default timezone that this instance of Nagios runs in. Useful if you have multiple instances of Nagios that need to run from the same server, but have different local times associated with them. If not specified, Nagios will use the system configured timezone.



Note: If you use this option to specify a custom timezone, you will also need to alter the Apache configuration directives for the CGIs to specify the timezone you want. Example:

```
<Directory "/usr/local/nagios/sbin/">
```

```
SetEnv TZ "US/Mountain"
```

```
...
```

</Directory>

表 5.112. 非法对象名字符

格式:	<code>illegal_object_name_chars=<chars...></code>
样例:	<code>illegal_object_name_chars=~!\$%^&*~ '<>?,()=</code>

This option allows you to specify illegal characters that cannot be used in host names, service descriptions, or names of other object types. Nagios will allow you to use most characters in object definitions, but I recommend not using the characters shown in the example above. Doing may give you problems in the web interface, notification commands, etc.

表 5.113. 非法宏输出字符

格式:	<code>illegal_macro_output_chars=<chars...></code>
样例:	<code>illegal_macro_output_chars=~\$^&~ '<></code>

This option allows you to specify illegal characters that should be stripped from [macros](#) before being used in notifications, event handlers, and other commands. This DOES NOT affect macros used in service or host check commands. You can choose to not strip out the characters shown in the example above, but I recommend you do not do this. Some of these characters are interpreted by the shell (i.e. the backtick) and can lead to security problems. The following macros are stripped of the characters you specify:

`$HOSTOUTPUT$, $HOSTPERFDATA$, $HOSTACKAUTHOR$, $HOSTACKCOMMENT$, $SERVICEOUTPUT$, $SERVICEPERFDATA$, $SERVICEACKAUTHOR$, and $SERVICEACKCOMMENT$`

表 5.114. 正则表达式选项

格式:	<code>use_regexp_matching=<0/1></code>
样例:	<code>use_regexp_matching=0</code>

This option determines whether or not various directives in your [对象定义](#) will be processed as regular expressions. More information on how this works can be found [here](#).

- 0 = Don't use regular expression matching (default)

- 1 = Use regular expression matching

表 5.115. True Regular Expression Matching Option

格式:	<code>use_true_regexp_matching=<0/1></code>
样例:	<code>use_true_regexp_matching=0</code>

If you've enabled regular expression matching of various object directives using the [use_regexp_matching](#) option, this option will determine when object directives are treated as regular expressions. If this option is disabled (the default), directives will only be treated as regular expressions if they contain *, ?, +, or \. If this option is enabled, all appropriate directives will be treated as regular expressions – be careful when enabling this! More information on how this works can be found [here](#).

- 0 = Don't use true regular expression matching (default)
- 1 = Use true regular expression matching

表 5.116. 管理员 EMail 帐号

格式:	<code>admin_email=<email_address></code>
样例:	<code>admin_email=root@localhost.localdomain</code>

This is the email address for the administrator of the local machine (i.e. the one that Nagios is running on). This value can be used in notification commands by using the \$ADMINEMAIL\$[macro](#).

表 5.117. 管理员 BP 机帐号

格式:	<code>admin_pager=<pager_number_or_pager_email_gateway></code>
样例:	<code>admin_pager=pageroot@localhost.localdomain</code>

This is the pager number (or pager email gateway) for the administrator of the local machine (i.e. the one that Nagios is running on). The pager number/address can be used in notification commands by using the \$ADMINPAGER\$[macro](#).

表 5.118. Event Broker Options

格式:	<code>event_broker_options=<#></code>
-----	---

样例:	<code>event_broker_options=-1</code>
-----	--------------------------------------

This option controls what (if any) data gets sent to the event broker and, in turn, to any loaded event broker modules. This is an advanced option. When in doubt, either broker nothing (if not using event broker modules) or broker everything (if using event broker modules). Possible values are shown below.

- 0 = Broker nothing
- -1 = Broker everything
- # = See BROKER_* definitions in source code (include/broker.h) for other values that can be OR'ed together

表 5.119. Event Broker Modules

格式:	<code>broker_module=<modulepath> [moduleargs]</code>
样例:	<code>broker_module=/usr/local/nagios/bin/ndomod.o</code> <code>cfg_file=/usr/local/nagios/etc/ndomod.cfg</code>

This directive is used to specify an event broker module that should be loaded by Nagios at startup. Use multiple directives if you want to load more than one module. Arguments that should be passed to the module at startup are separated from the module path by a space.

!!! WARNING !!!

Do NOT overwrite modules while they are being used by Nagios or Nagios will crash in a fiery display of SEGFAULT glory. This is a bug/limitation either in dlopen(), the kernel, and/or the filesystem. And maybe Nagios...

The correct/safe way of updating a module is by using one of these methods:

- Shutdown Nagios, replace the module file, restart Nagios
- While Nagios is running... delete the original module file, move the new module file into place, restart Nagios

表 5.120. 调试文件

格式:	<code>debug_file=<file_name></code>
样例:	<code>debug_file=/usr/local/nagios/var/nagios.debug</code>

This option determines where Nagios should write debugging information. What (if any) information is written is determined by the [debug_level](#) and [debug_verbosity](#) options. You can have Nagios automatically rotate the debug file when it reaches a certain size by using the [max_debug_file_size](#) option.

表 5.121. 调试等级

格式:	<code>debug_level=<#></code>
样例:	<code>debug_level=24</code>

该选项决定Nagios将往 [debug_file](#)文件里写入什么调试信息。下面值是可以逻辑或关系:

- -1 = Log everything
- 0 = Log nothing (default)
- 1 = Function enter/exit information
- 2 = Config information
- 4 = Process information
- 8 = Scheduled event information
- 16 = Host/service check information
- 32 = Notification information
- 64 = Event broker information

表 5.122. Debug Verbosity

格式:	<code>debug_verbosity=<#></code>
样例:	<code>debug_verbosity=1</code>

This option determines how much debugging information Nagios should write to the [debug_file](#) debug_file.

- 0 = Basic information
- 1 = More detailed information (default)
- 2 = Highly detailed information

表 5.123. 调试文件最大长度

格式:	<code>max_debug_file_size=<#></code>
样例:	<code>max_debug_file_size=1000000</code>

该选项定义了以字节为单位的 [debug_file](#) 调试文件最大长度。如果文件增至大于该值，将会自动被命名为 .old 扩展名的文件，如果 .old 扩展名已经存在，那么旧 .old 文件将被删除。这可以保证在 Nagios 调试时磁盘空间不会过多占用而失控。

5.3. 对象配置概览

5.3.1. 什么是对象？

对象是指所有在监控和通知逻辑中涉及到的元素。对象的类型包括：

- 服务
- 服务组
- 主机
- 主机组
- 联系人
- 联系人组
- 命令
- 时间周期
- 通知扩展
- 通知和执行依赖关系

更多有关对象和它们之间关系的说明见下面。

5.3.2. 对象在哪里定义？

对象可以在一个配置文件 [cfg_file](#) 或是多个由主配置文件对象保存目录 [cfg_dir](#) 里配置文件来定义。
提示

当按照 [快速安装指南](#) 进行安装后，几个对象配置文件的样例放在了 `/usr/local/nagios/etc/objects/` 目录下。可以用这些样例文件来搞清楚对象继承关系并学习如何进行自己的对象定义。

5.3.3. 对象如何定义？

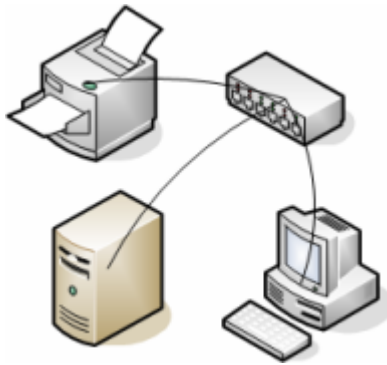
对象可以在一个用柔性化模板样式来定义，模板可使得对 Nagios 的配置管理更为容易，有关如何进行对象定义的基本信息可以查阅 [这篇文件](#)。

一旦熟悉了如何进行对象定义的基础，需要阅读 [对象继承](#) 以在将来应用中配置更为鲁棒（就是尽量使用对象继承关系啦）。经验丰富的使用者可以在 [对象定义诀窍](#) 一文中发掘到一些有关对象定义的高级特性。

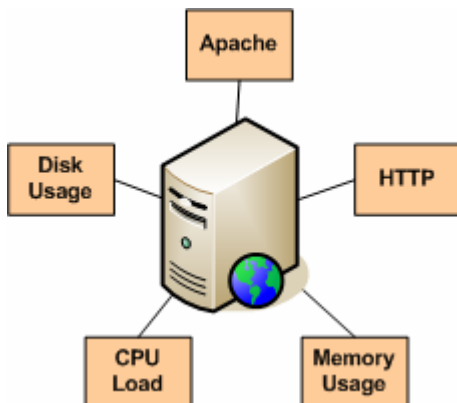
关于对象的解释

下面在一些主要的对象的解释...

- [主机](#)是监控逻辑中的核心对象之一。主机的重要属性有：
 - 主机通常在网络中的物理设备(如服务器、工作站、路由器、交换机和打印机等)；
 - 主机有某种形式的地址(象 IP 或 MAC 地址)；
 - 主机有一个或多个绑定的服务；
 - 主机与其他的主机间可以有父/子节点的关系，通常反应出真实世界里的网络联接关系，而联接关系会在 [网络可达性](#)逻辑中用到。
- [主机组](#)是一台或多台主机组成的组。主机成组可以如下工作更简单(1)在Nagios的Web接口里查看相关的主机状态(2)使用 [对象定义诀窍](#)来简化配置。



- [服务](#)监控逻辑中的一个核心对象之一。在主机上的服务用户可以：
 - 主机的属性(CPU 负荷、磁盘利用率、启动时间等)；
 - 主机提供的服务(HTTP, POP3, FTP, SSH 等等)；
 - 其他与主机有关的信息(DNS 记录等)；
- [服务组](#)是一个或多个服务组成的组。服务组可以对如下工作更简单(1)在Nagios的Web接口里查看相关的服务状态(2)使用 [对象定义诀窍](#)来简化配置。



- [联系人](#)是那些涉及到通知过程中的人：
 - 有多种通知联系人的方法(对讲机、BP 机、EMail、即时信息等)；

- 联系人收到的通知来自于其负责的主机或服务；
- [联系人组](#)是一个或多个联系人组成的组。联系人组可以简化在主机或服务故障时负责的人员划分。



- [时间周期](#)用于控制：
- 主机或服务被监控的时间；
- 联系人可接收通知的时间；

时间段时如何工作的信息可以查阅 [这篇文档](#)。



- [命令](#)是指出Nagios用哪个程序、脚本等，它必须可执行后完成：
- 主机和服务检测
- 通知
- 事件处理
- 和其他...



5.4. CGI 配置文件选项

注意

当创建或编辑配置文件时，要遵守如下要求：

- 以符号'#'开头的行将视为注释不做处理；
- 变量必须是新起的一行 — 变量之前不能有空格符；
- 变量名是大小写敏感的；

5.4.1. 样例配置文件

提示



一个CGI的样例配置文件(/usr/local/nagios/etc/cgi.cfg)已经安装到位，如果你是按照[快速安装指南](#)来操作的话。

5.4.2. 配置文件的位置

默认情况下，Nagios期望的CGI配置文件被命名为cgi.cfg并且该配置文件被放在了[主配置文件](#)指定的位置。如果你想改变名称和位置，你可以在Apache里配置一个环境变量叫做NAGIO_CGI_CONFIG的(里面设置好文件名和位置)给CGI程序用。如何来做可以查看Apache文档里的说明。

5.4.3. 配置文件里的变量

下面将给出每个主配置文件里的变量与值选项说明...

表 5.124. 主配置文件的位置

格式:	<code>main_config_file=<file_name></code>
举例:	<code>main_config_file=/usr/local/nagios/etc/nagios.cfg</code>

它用于指向[主配置文件](#)所在的位置。CGI模块需要知道在哪里可以得到主配置文件以取得配置信息、当前的主机和服务的状态等。

表 5.125. HTML 文件的系统路径

格式:	<code>physical_html_path=<path></code>
举例:	<code>physical_html_path=/usr/local/nagios/share</code>

它用于指明用于服务器或工作站上的HTML文件所在的**系统**路径。Nagios假定文档和图片文件被分别放在了docs/和images/两个子目录下。

表 5.126. URL 里的HTML路径

格式:	<code>url_html_path=<path></code>
举例:	<code>url_html_path=/nagios</code>

如果通过Web浏览器来操作Nagios，你要通过一个URL如http://www.myhost.com/nagios来操作的话，则需要设置为/nagios。一般是用这个URL来操作Nagios的HTML页面。

表 5.127. 应用认证

格式:	<code>use_authentication=<0/1></code>
-----	---

举例:	<code>use_authentication=1</code>
-----	-----------------------------------

该选项控制着CGI模块里，对于用户操作或是取得信息时是否需要打开认证和授权功能。如果你断定你不使用认证，一定要把 [CGI命令](#) 移走以免没有授权的用户发出Nagios命令。如果不使用认证功能，CGI模块不会向Nagios发出命令，但我同时也建议你也将CGI模块同时移到安全位置。更多的有关设置认证与授权的内容可以查看 [这个](#) 文件。

- 0 = 不使用认证功能
- 1 = 使用认证与授权功能(默认值)

表 5.128. 默认用户名

格式:	<code>default_user_name=<username></code>
举例:	<code>default_user_name=guest</code>

用这个变量可以设置一个默认的用户来操作 CGI 程序。它可以在一个加密的域里(如在防火墙后建立的WEB)不需要WEB认证就可以操作CGI模块。你可能需要这个功能来避免仅仅在一个非加密的服务器上(通过因特网以明文方式来传递你的口令)来做基本的认证。

Important:除非你是在一个加密的WEB服务器上并且保证每个进入该域的用户都具备CGI操作权，否则的话，你不要定义这个默认用户。如果你决定用它，那么任何一个未经认证的WEB服务器用户都可以继承你设定的全部权限！

表 5.129. 系统和进程的信息操作权

格式:	<code>authorized_for_system_information=<user1>,<user2>,<user3>,...<usern></code>
举例:	<code>authorized_for_system_information=nagiosadmin,theboss</code>

这是一个以逗号分隔的列表，列举出了在 [扩展CGI信息](#) 里查看系统和进程信息的可认证用户。在列表中列出的用户并不会自动被授权可发出系统和进程的命令。如果你想也同时可以发出系统和进程命令，你必须把这些用户也加到 [authorized_for_system_commands](#) 变量之中。更多的如何给CGI模块设置认证和配置授权的内容可以查阅 [这个](#) 文档。

表 5.130. 系统和进程的命令操作权

格式:	<code>authorized_for_system_commands=<user1>,<user2>,<user3>,...<usern></code>
-----	--

举例:	<code>authorized_for_system_commands=nagiosadmin</code>
-----	---

这是一个以逗号分隔的列表，列出了可以通过 [CGI命令](#)发出系统和进程命令的**被认证用户**。在列表中的用户并**没有**被自动授权查看系统和进程的信息。如果你想让用户也同时可以查看系统和进程信息的话，你必须把这些用户也加到 [authorized for system information](#)变量里面。更多的如何给CGI模块设置认证和配置授权的内容可以查阅 [这个](#)文档。

表 5.131. 配置的信息获取权限

格式:	<code>authorized_for_configuration_information=<user1>,<user2>,<user3>,...<usern></code>
举例:	<code>authorized_for_configuration_information=nagiosadmin</code>

这是一个以逗号分隔的列表，列出了可以通过 [配置查看CGI](#)里查看配置信息的**可认证用户**。这些列表中的用户可以查看全部的配置好的主机、主机组、服务、联系人、联系人组等的配置信息。更多的如何给CGI模块设置认证和配置授权的内容可以查阅 [这个](#)文档。

表 5.132. 全局主机的信息获取权限

格式:	<code>authorized_for_all_hosts=<user1>,<user2>,<user3>,...<usern></code>
举例:	<code>authorized_for_all_hosts=nagiosadmin,theboss</code>

这是一个以逗号分隔的列表，列出了可以查看全部主机的状态和配置信息的**被认证用户**。这些列表中的用户同时被授权查看在全部的服务信息。但列表中的用户并**没有**自动地授权向全部的主机或服务发出命令。如果你想让这些用户同时可以向全部主机和服务发出命令，你必须将用户加入到 [authorized for all host commands](#)变量里。更多的如何给CGI模块设置认证和配置授权的内容可以查阅 [这个](#)文档。

表 5.133. 全局主机的命令操作权

格式:	<code>authorized_for_all_host_commands=<user1>,<user2>,<user3>,...<usern></code>
举例:	<code>authorized_for_all_host_commands=nagiosadmin</code>

这是一个以逗号分隔的列表，列出了可以通过 [命令CGI](#)功能模块向全部主机发出命令的**被授权用户**。列表中的用户同时自动地被授权可以向全部服务发出命令。但列表中的用户并**没有**自动地授权可以查看全

部的主机或服务的状态和配置信息，如果你想让用户同样可以查看状态和配置信息，你需要将用户加入到 [authorized_for_all_hosts](#) 变量之中。更多的如何给CGI模块设置认证和配置授权的内容可以查阅 [这个文档](#)。

表 5.134. 全局服务的信息获取权

格式:	<code>authorized_for_all_services=<user1>,<user2>,<user3>,...<usern></code>
举例:	<code>authorized_for_all_services=nagiosadmin,theboss</code>

这是一个以逗号分隔的列表，列出了可以查看全部服务的状态和配置的被授权用户。但列表中的用户并没有自动地授权可以查看全部主机的信息。列表中的用户并没有自动地授权向全部服务发送命令。如果你想让这些用户也同样可以发全部服务发送命令，你必须将这些用户加入到 [authorized_for_all_service_commands](#) 变量之中。更多的如何给CGI模块设置认证和配置授权的内容可以查阅 [这个文档](#)。

表 5.135. 全局服务的命令操作权

格式:	<code>authorized_for_all_service_commands=<user1>,<user2>,<user3>,...<usern></code>
举例:	<code>authorized_for_all_service_commands=nagiosadmin</code>

这是一个以逗号分隔的列表，列出了可以通过 [命令CGI](#)来向全部服务发送命令的被授权用户。但列表中的用户并没有自动地授权向全部主机发送命令。列表中的用户也没有自动地授权查看全部主机的状态和配置信息。如果你想让这些用户同样可以查年全部服务的状态和服务的信息，你必须把这些用户加入到 [authorized_for_all_services](#) 变量中。更多的如何给CGI模块设置认证和配置授权的内容可以查阅 [这个文档](#)。

表 5.136. 锁定动作者的用户名

格式:	<code>lock_author_names=[0/1]</code>
举例:	<code>lock_author_names=1</code>

该选项将使用 WEB 接口时在提交注释、做内容确认和制订宕机计划等操作时限制修改已经他们的动作提交者的名字。如果该选项使能，那么用户在做这些进行命令时将不能修改发出操作者的名字。

- 0 = 允许用户在提交命令时修改名字

- 1 = 不许用户提交命令时修改名字(默认值)

表 5.137. 网络拓扑图的背景图设置

格式:	<code>statusmap_background_image=<image_file></code>
举例:	<code>statusmap_background_image=smbbackground.gd2</code>

该选项将让你可以在使用 [网络拓扑图](#) 时可以指定一个图形文件做为背景图，如果你选择了使用用户定义坐标来绘制的二维网络拓扑图的话。该背景图文件将不能为其他绘制方式提供背景。它假定这个文件是放在图像文件的路径里了(如/usr/local/nagios/share/images)。该路径将自动地在 [physical_html_path](#) 域之后加上"/images"生成路径。注意，这个图像文件的格式可以是GIF、JPEG、PNG 或GD2 格式。而推荐是GD2 格式的文件，因为它可以在生成二维图时降低CPU负荷。

表 5.138. 默认的二维拓扑图层绘制方式

格式:	<code>default_statusmap_layout=<layout_number></code>
举例:	<code>default_statusmap_layout=4</code>

这个选项将让你指定出 [网络拓扑图CGI](#) 的默认绘制方式，可用的选项值有：

表 5.139. Statusmap 的<layout_number>取值

Value	Layout Method
0	用户定义坐标系
1	深度图
2	树形折叠图
3	平衡权图
4	圆形图
5	圆形图(出标记的)
6	圆形图(气泡式)

表 5.140. 三维空间的容纳器

格式:	<code>statuswrl_include=<vrml_file></code>
举例:	<code>statuswrl_include=myworld.wrl</code>

这个选项将让你指定一个你的对象实体在哪个三维空间的容器里展现。它默认是文件已经存放在指定的路径下了,该路径由 [physical_html_path](#)域来指定。注意,这个文件必须是合格的虚拟现实建模(VRML)文件(如你可以在它的专用浏览器里可以查看它)。

表 5.141. 默认三维空间坐标生成算法

格式:	<code>default_statuswrl_layout=<layout_number></code>
举例:	<code>default_statuswrl_layout=4</code>

该选项让你指定在 [三维空间图](#)里对象的三维空间坐标的生成算法。可用的选项值有:

表 5.142. Statuswrl 的<layout_number>取值

值	绘制算法
0	用户定义坐标系
2	折叠树
3	平衡树
4	圆形

表 5.143. CGI 模块的刷新速率

格式:	<code>refresh_rate=<rate_in_seconds></code>
举例:	<code>refresh_rate=90</code>

该选项将让你指定以秒为单位的对于CGI模块刷新的周期,CGI模块有 [状态列表](#)、[二维拓扑图](#)和 [扩展信息](#)等CGI模块。

表 5.144. 声音报警

格式:	<code>host_unreachable_sound=<sound_file></code>
-----	--

	<p>host_down_sound=<sound_file></p> <p>service_critical_sound=<sound_file></p> <p>service_warning_sound=<sound_file></p> <p>service_unknown_sound=<sound_file></p>
举例:	<p>host_unreachable_sound=hostu.wav</p> <p>host_down_sound=hostd.wav</p> <p>service_critical_sound=critical.wav</p> <p>service_warning_sound=warning.wav</p> <p>service_unknown_sound=unknown.wav</p>

这个选项将让你指定在查看 [状态列表](#) 时如果有故障发生，你的浏览器里将发出哪个声音文件。如果有故障将按指定的临界故障类型来播放不同的声音文件。这些临界的故障类型是一个或多个主机不可达，至少是一个或多个服务处于未知的状态(见上例中的次序)。声音文件将假定你放在了 [HTML目录](#) 的“media/”子目录里(如/usr/local/nagios/share/media)。

表 5.145. Ping 语法

格式:	ping_syntax=<command>
举例:	ping_syntax=/bin/ping -n -U -c 5 \$HOSTADDRESS\$

这个选项给出了当从WAP接口(使用 [statuswml CGI](#))做PING一个主机操作时的PING的语法。你必须给出包含全路径名的PING的执行文件及全部参数的命令行。命令中使用\$HOSTADDRESS\$宏来预指定在命令执行前对哪个地址替换并执行PING检测。

表 5.146. 扩展 HTML 标记选项

格式:	escape_html_tags=[0/1]
举例:	escape_html_tags=1

这个选项将决定是否在主机和服务(插件)的检测输出中包含使用 HTML 的扩展选项。如果你使能了它，你的插件将不能使用可点击的超链接标记。

表 5.147. 注释的 URL 指向

格式:	<code>notes_url_target=[target]</code>
举例:	<code>notes_url_target=_blank</code>

这个选项决定了你的注释 URL 必须要显示的 URL 目标。合法的选项内容包括 `_blank`、`_self`、`_top`、`_parent` 或是其他合法目标的名字。

表 5.148. 动作的 URL 指向

格式:	<code>action_url_target=[target]</code>
举例:	<code>action_url_target=_blank</code>

这个选项给定了框内对象的动作里显示的动作 URL 的目标。合法的选项值包括 `_blank`、`_self`、`_top`、`_parent` 或是任何其他合法目标名字。

表 5.149. Splunk 集成选项

格式:	<code>enable_splunk_integration=[0/1]</code>
举例:	<code>enable_splunk_integration=1</code>

这个选项决定了在WEB接口里与Splunk集成功能是否集成。如果使能它，你页面中将在许多地方呈现出“Splunk It”的链接，CGI模块页面(日志文件、告警历史、主机和服务的详细信息等)里都有。如果你想对特别的故障发生想知道原委时很有用。更多关于Splunk的信息请访问 <http://www.splunk.com/>。

表 5.150. Splunk URL

格式:	<code>splunk_url=<path></code>
举例:	<code>splunk_url=http://127.0.0.1:8000/</code>

这个选项设置了指向Splunk网站的URL。在 [enable splunk integration](#)使能时这个URL被CGI模块用于指向Splunk。

第 6 章 Nagios 监控与配置的基本概念

6.1. 对象定义

6.1.1. 介绍

Nagios对象格式的一个特点是可以创建上下继承关系的对象定义。一个如何实现对象继承关系的解释可查阅 [这篇文档](#)。强烈建议你在阅读过下面内容后要再熟悉一下继承关系，因为它将使对象定义创建和维护变得更为容易，同样，还得阅读 [对象定义诀窍](#) 一文以使一些冗长定义任务变得简短。

注意

当创建或编辑配置文件时，要遵守如下要求：

- 以符号'#'开头的行将视为注释不做处理；
- 变量名是大小写敏感的；

6.1.2. 注意状态保持设置

需要着重指出一点，当修改了配置文件时有几个在主机、服务和联系人定义里的域值不会清除。有这种特性的对象域在下面被标记了星号(*)。这个原因是由于Nagios会将一些对象域值会保存在 [状态保持文件](#)里的值来覆盖配置文件，前提是配置了对程序内容全面地 [状态保持](#)选项使能并且域里的值在运行时被 [外部命令](#)修改过。

绕过这个问题的一个方法是将非状态信息的保持选项关闭掉，在主机、服务和联系人对象定义里用 `retain_nonstatus_information` 选项开关。关掉这个选项后会令 Nagios 在重启动时使用配置文件里给出的域值而不是从状态保持文件中取值。

6.1.3. 样例配置文件

注意

如果按照 [快速安装指南](#)来操作的话，一个样例对象配置文件将被安装到/usr/local/nagios/etc/目录里。

6.1.4. 对象种类

- [第 6.1.5 节 “主机定义 ”](#)
- [第 6.1.6 节 “ 主机组定义 ”](#)
- [第 6.1.7 节 “ 服务定义 ”](#)
- [第 6.1.8 节 “ 服务组定义 ”](#)
- [第 6.1.9 节 “ 联系人定义 ”](#)
- [第 6.1.10 节 “ 联系人组定义 ”](#)
- [第 6.1.11 节 “ 时间周期定义 ”](#)
- [第 6.1.12 节 “ 命令定义 ”](#)
- [第 6.1.13 节 “ 服务依赖定义 ”](#)
- [第 6.1.14 节 “ 服务扩展定义 ”](#)

- [第 6.1.15 节 “ 主机依赖定义 ”](#)
- [第 6.1.16 节 “ 主机扩展定义 ”](#)
- [第 6.1.17 节 “ 额外主机信息定义 ”](#)
- [第 6.1.18 节 “ 额外服务信息定义 ”](#)

6.1.5. 主机定义

描述：

主机被定义为存在于网络中的一个物理服务器、工作站或设备等。

定义格式：

注意

标记了(*)的域是必备的而黑色是可选的。

```
define host{
    host_name      host_name(*)
    alias          alias(*)
    display_name    display_name
    address         address(*)
    parents         host_names
    hostgroups      hostgroup_names
    check_command    command_name
    initial_state    [o, d, u]
    max_check_attempts    #(*)
    check_interval    #
    retry_interval    #
    active_checks_enabled    [0/1]
    passive_checks_enabled    [0/1]
    check_period      timeperiod_name(*)
    obsess_over_host    [0/1]
    check_freshness    [0/1]
    freshness_threshold    #
    event_handler      command_name
    event_handler_enabled    [0/1]
    low_flap_threshold    #
```

```

high_flap_threshold      #
flap_detection_enabled   [0/1]
flap_detection_options   [o, d, u]
process_perf_data       [0/1]
retain_status_information [0/1]
retain_nonstatus_information [0/1]
contacts                contacts(*)
contact_groups          contact_groups(*)
notification_interval    #(*)
first_notification_delay  #
notification_period      timeperiod_name(*)
notification_options     [d, u, r, f, s]
notifications_enabled    [0/1]
stalking_options        [o, d, u]
notes                   note_string
notes_url               url
action_url              url
icon_image              image_file
icon_image_alt          alt_string
vrml_image              image_file
statusmap_image         image_file
2d_coords               x_coord, y_coord
3d_coords               x_coord, y_coord, z_coord
...
}

```

定义样例:

```

define host{
    host_name          bogus-router
    alias              Bogus Router #1
    address             192. 168. 1. 254

```

```

        parents                server-backbone
        check_command           check-host-alive
        check_interval          5
        retry_interval          1
        max_check_attempts      5
        check_period            24x7
        process_perf_data       0
        retain_nonstatus_information 0
        contact_groups          router-admins
        notification_interval    30
        notification_period     24x7
        notification_options    d,u,r
    }

```

域描述:

host_name: This directive is used to define a short name used to identify the host. It is used in host group and service definitions to reference this particular host. Hosts can have multiple services (which are monitored) associated with them. When used properly, the `$HOSTNAME$` [macro](#) will contain this short name.

alias: This directive is used to define a longer name or description used to identify the host. It is provided in order to allow you to more easily identify a particular host. When used properly, the `$HOSTALIAS$` [macro](#) will contain this alias/description.

address: This directive is used to define the address of the host. Normally, this is an IP address, although it could really be anything you want (so long as it can be used to check the status of the host). You can use a FQDN to identify the host instead of an IP address, but if DNS services are not available this could cause problems. When used properly, the `$HOSTADDRESS$` [macro](#) will contain this address. Note: If you do not specify an address directive in a host definition, the name of the host will be used as its address. A word of caution about doing this, however – if DNS fails, most of your service checks will fail because the plugins will be unable to resolve the host name.

display_name: This directive is used to define an alternate name that should be displayed in the web interface for this host. If not specified, this defaults to the value you specify for the **host_name** directive. Note: The current CGIs do not use this option, although future versions of the web interface will.

parents: This directive is used to define a comma-delimited list of short names of the "parent" hosts for this particular host. Parent hosts are typically routers, switches, firewalls, etc. that lie between the monitoring host and a remote hosts. A router, switch, etc. which is closest to the remote host is considered to be that host's "parent". Read the "Determining Status and Reachability of Network Hosts" document located [here](#) for more information. If this host is on the same network segment as the host doing the monitoring (without any intermediate routers, etc.) the host is considered to be on the local network and will not have a parent host. Leave this value blank if the host does not have a parent host (i.e. it is on the same segment as the Nagios host). The order in which you specify parent hosts has no effect on how things are monitored.

hostgroups: This directive is used to identify the **short name(s)** of the [hostgroup\(s\)](#) that the host belongs to. Multiple hostgroups should be separated by commas. This directive may be used as an alternative to (or in addition to) using the **members** directive in [hostgroup definitions](#).

check_command: This directive is used to specify the **short name** of the [command](#) that should be used to check if the host is up or down. Typically, this command would try and ping the host to see if it is "alive". The command must return a status of OK (0) or Nagios will assume the host is down. If you leave this argument blank, the host will **not** be actively checked. Thus, Nagios will likely always assume the host is up (it may show up as being in a "PENDING" state in the web interface). This is useful if you are monitoring printers or other devices that are frequently turned off. The maximum amount of time that the notification command can run is controlled by the [host_check_timeout](#) option.

initial_state: By default Nagios will assume that all hosts are in UP states when in starts. You can override the initial state for a host by using this directive. Valid options are: o = UP, d = DOWN, and u = UNREACHABLE.

max_check_attempts: This directive is used to define the number of times that Nagios will retry the host check command if it returns any state other than an OK state. Setting this value to 1 will cause Nagios to generate an alert without retrying the host check again. Note: If you

do not want to check the status of the host, you must still set this to a minimum value of 1. To bypass the host check, just leave the **check_command** option blank.

check_interval: This directive is used to define the number of "time units" between regularly scheduled checks of the host. Unless you've changed the [interval_length](#) directive from the default value of 60, this number will mean minutes. More information on this value can be found in the [check scheduling](#) documentation.

retry_interval: This directive is used to define the number of "time units" to wait before scheduling a re-check of the hosts. Hosts are rescheduled at the retry interval when they have changed to a non-UP state. Once the host has been retried `max_attempts` times without a change in its status, it will revert to being scheduled at its "normal" rate as defined by the `check_interval` value. Unless you've changed the [interval_length](#) directive from the default value of 60, this number will mean minutes. More information on this value can be found in the [check scheduling](#) documentation.

active_checks_enabled **: This directive is used to determine whether or not active checks (either regularly scheduled or on-demand) of this host are enabled. Values: 0 = disable active host checks, 1 = enable active host checks.

passive_checks_enabled **: This directive is used to determine whether or not passive checks are enabled for this host. Values: 0 = disable passive host checks, 1 = enable passive host checks.

check_period: This directive is used to specify the short name of the [time period](#) during which active checks of this host can be made.

obsess_over_host **: This directive determines whether or not checks for the host will be "obsessed" over using the [ochp command](#).

check_freshness **: This directive is used to determine whether or not [freshness checks](#) are enabled for this host. Values: 0 = disable freshness checks, 1 = enable freshness checks.

freshness_threshold: This directive is used to specify the freshness threshold (in seconds) for this host. If you set this directive to a value of 0, Nagios will determine a freshness threshold to use automatically.

event_handler: This directive is used to specify the **short name** of the [command](#) that should be run whenever a change in the state of the host is detected (i.e. whenever it goes down or recovers). Read the documentation on [event handlers](#) for a more detailed explanation of how to

write scripts for handling events. The maximum amount of time that the event handler command can run is controlled by the [event_handler_timeout](#) option.

event_handler_enabled [](#):** This directive is used to determine whether or not the event handler for this host is enabled. Values: 0 = disable host event handler, 1 = enable host event handler.

low_flap_threshold: This directive is used to specify the low state change threshold used in flap detection for this host. More information on flap detection can be found [here](#). If you set this directive to a value of 0, the program-wide value specified by the [low host flap threshold](#) directive will be used.

high_flap_threshold: This directive is used to specify the high state change threshold used in flap detection for this host. More information on flap detection can be found [here](#). If you set this directive to a value of 0, the program-wide value specified by the [high host flap threshold](#) directive will be used.

flap_detection_enabled [](#):** This directive is used to determine whether or not flap detection is enabled for this host. More information on flap detection can be found [here](#). Values: 0 = disable host flap detection, 1 = enable host flap detection.

flap_detection_options: This directive is used to determine what host states the [flap detection logic](#) will use for this host. Valid options are a combination of one or more of the following: o = UP states, d = DOWN states, u = UNREACHABLE states.

process_perf_data [](#):** This directive is used to determine whether or not the processing of performance data is enabled for this host. Values: 0 = disable performance data processing, 1 = enable performance data processing.

retain_status_information: This directive is used to determine whether or not status-related information about the host is retained across program restarts. This is only useful if you have enabled state retention using the [retain state information](#) directive. Value: 0 = disable status information retention, 1 = enable status information retention.

retain_nonstatus_information: This directive is used to determine whether or not non-status information about the host is retained across program restarts. This is only useful if you have enabled state retention using the [retain state information](#) directive. Value: 0 = disable non-status information retention, 1 = enable non-status information retention.

contacts: This is a list of the **short names** of the [contacts](#) that should be notified whenever there are problems (or recoveries) with this host. Multiple contacts should be separated by commas. Useful if you want notifications to go to just a few people and don't want to configure [contact groups](#). You must specify at least one contact or contact group in each host definition.

contact_groups: This is a list of the **short names** of the [contact groups](#) that should be notified whenever there are problems (or recoveries) with this host. Multiple contact groups should be separated by commas. You must specify at least one contact or contact group in each host definition.

notification_interval: This directive is used to define the number of "time units" to wait before re-notifying a contact that this server is **still** down or unreachable. Unless you've changed the [interval length](#) directive from the default value of 60, this number will mean minutes. If you set this value to 0, Nagios will **not** re-notify contacts about problems for this host - only one problem notification will be sent out.

first_notification_delay: This directive is used to define the number of "time units" to wait before sending out the first problem notification when this host enters a non-UP state. Unless you've changed the [interval length](#) directive from the default value of 60, this number will mean minutes. If you set this value to 0, Nagios will start sending out notifications immediately.

notification_period: This directive is used to specify the short name of the [time period](#) during which notifications of events for this host can be sent out to contacts. If a host goes down, becomes unreachable, or recovers during a time which is not covered by the time period, no notifications will be sent out.

notification_options: This directive is used to determine when notifications for the host should be sent out. Valid options are a combination of one or more of the following: d = send notifications on a DOWN state, u = send notifications on an UNREACHABLE state, r = send notifications on recoveries (OK state), f = send notifications when the host starts and stops [flapping](#), and s = send notifications when [scheduled downtime](#) starts and ends. If you specify n (none) as an option, no host notifications will be sent out. If you do not specify any notification options, Nagios will assume that you want notifications to be sent out for all possible states. Example: If you specify d,r in this field, notifications will only be sent out when the host goes DOWN and when it recovers from a DOWN state.

notifications_enabled ******: This directive is used to determine whether or not notifications for this host are enabled. Values: 0 = disable host notifications, 1 = enable host notifications.

stalking_options: This directive determines which host states "stalking" is enabled for. Valid options are a combination of one or more of the following: o = stalk on UP states, d = stalk on DOWN states, and u = stalk on UNREACHABLE states. More information on state stalking can be found [here](#).

notes: This directive is used to define an optional string of notes pertaining to the host. If you specify a note here, you will see the it in the [extended information](#) CGI (when you are viewing information about the specified host).

notes_url: This variable is used to define an optional URL that can be used to provide more information about the host. If you specify an URL, you will see a red folder icon in the CGIs (when you are viewing host information) that links to the URL you specify here. Any valid URL can be used. If you plan on using relative paths, the base path will be the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`). This can be very useful if you want to make detailed information on the host, emergency contact methods, etc. available to other support staff.

action_url: This directive is used to define an optional URL that can be used to provide more actions to be performed on the host. If you specify an URL, you will see a red "splat" icon in the CGIs (when you are viewing host information) that links to the URL you specify here. Any valid URL can be used. If you plan on using relative paths, the base path will be the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`).

icon_image: This variable is used to define the name of a GIF, PNG, or JPG image that should be associated with this host. This image will be displayed in the various places in the CGIs. The image will look best if it is 40x40 pixels in size. Images for hosts are assumed to be in the `logos/` subdirectory in your HTML images directory (i.e. `/usr/local/nagios/share/images/logos`).

icon_image_alt: This variable is used to define an optional string that is used in the ALT tag of the image specified by the `<icon_image>` argument.

vrml_image: This variable is used to define the name of a GIF, PNG, or JPG image that should be associated with this host. This image will be used as the texture map for the specified host in the [statuswrl](#) CGI. Unlike the image you use for the `<icon_image>` variable, this one should

probably **not** have any transparency. If it does, the host object will look a bit wierd. Images for hosts are assumed to be in the `logos/` subdirectory in your HTML images directory (i.e. `/usr/local/nagios/share/images/logos`).

statusmap_image: This variable is used to define the name of an image that should be associated with this host in the [statusmap](#) CGI. You can specify a JPEG, PNG, and GIF image if you want, although I would strongly suggest using a GD2 format image, as other image formats will result in a lot of wasted CPU time when the statusmap image is generated. GD2 images can be created from PNG images by using the `pngtogd2` utility supplied with Thomas Boutell's [gd library](#). The GD2 images should be created in **uncompressed** format in order to minimize CPU load when the statusmap CGI is generating the network map image. The image will look best if it is 40x40 pixels in size. You can leave these option blank if you are not using the statusmap CGI. Images for hosts are assumed to be in the `logos/` subdirectory in your HTML images directory (i.e. `/usr/local/nagios/share/images/logos`).

2d_coords: This variable is used to define coordinates to use when drawing the host in the [statusmap](#) CGI. Coordinates should be given in positive integers, as the correspond to physical pixels in the generated image. The origin for drawing (0,0) is in the upper left hand corner of the image and extends in the positive x direction (to the right) along the top of the image and in the positive y direction (down) along the left hand side of the image. For reference, the size of the icons drawn is usually about 40x40 pixels (text takes a little extra space). The coordinates you specify here are for the upper left hand corner of the host icon that is drawn. Note: Don't worry about what the maximum x and y coordinates that you can use are. The CGI will automatically calculate the maximum dimensions of the image it creates based on the largest x and y coordinates you specify.

3d_coords: This variable is used to define coordinates to use when drawing the host in the [statuswrl](#) CGI. Coordinates can be positive or negative real numbers. The origin for drawing is (0.0,0.0,0.0). For reference, the size of the host cubes drawn is 0.5 units on each side (text takes a little more space). The coordinates you specify here are used as the center of the host cube.

6.1.6. 主机组定义

描述:

主机组是指一台或多台主机构成的组, 可使配置更简单或是为完成特定目的而在 [CGI](#) 里显示使用。

定义格式:

注意

标记了(*)的域是必备的而黑色是可选的。

```
define hostgroup{
    hostgroup_name      hostgroup_name(*)
    alias               alias(*)
    members             hosts
    hostgroup_members   hostgroups
    notes               note_string
    notes_url           url
    action_url          url
    ...
}
```

定义样例:

```
define hostgroup{
    hostgroup_name      novell-servers
    alias               Novell Servers
    members             netware1, netware2, netware3, netware4
}
```

域描述:

hostgroup_name: This directive is used to define a short name used to identify the host group.

alias: This directive is used to define is a longer name or description used to identify the host group. It is provided in order to allow you to more easily identify a particular host group.

members: This is a list of the **short names** of [hosts](#) that should be included in this group. Multiple host names should be separated by commas. This directive may be used as an alternative to (or in addition to) the **hostgroups** directive in [host definitions](#).

hostgroup_members: This optional directive can be used to include hosts from other "sub" host groups in this host group. Specify a comma-delimited list of short names of other host groups whose members should be included in this group.

notes: This directive is used to define an optional string of notes pertaining to the host. If you specify a note here, you will see the it in the [extended information](#) CGI (when you are viewing information about the specified host).

notes_url: This variable is used to define an optional URL that can be used to provide more information about the host group. If you specify an URL, you will see a red folder icon in the CGIs (when you are viewing hostgroup information) that links to the URL you specify here. Any valid URL can be used. If you plan on using relative paths, the base path will the the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`). This can be very useful if you want to make detailed information on the host group, emergency contact methods, etc. available to other support staff.

action_url: This directive is used to define an optional URL that can be used to provide more actions to be performed on the host group. If you specify an URL, you will see a red "splat" icon in the CGIs (when you are viewing hostgroup information) that links to the URL you specify here. Any valid URL can be used. If you plan on using relative paths, the base path will the the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`).

6.1.7. 服务定义

描述:

服务定义为在主机上运行的某种“应用服务”。这种服务定义得非常宽泛，可以是在主机上实际的服务进程 (POP3、SMTP、HTTP 等) 或是与主机有关的某种计量值 (PING 响应值、在线用户数、磁盘空闲空间等)，其中的差异见下面的说明。

定义格式:

注意

标记了^(*)的域是必备的而黑色是可选的。

```
define service{
    host_name          host_name(*)
    hostgroup_name      hostgroup_name
    service_description service_description(*)
    display_name        display_name
    servicegroups        servicegroup_names
    is_volatile          [0/1]
    check_command        command_name(*)
```

initial_state	[o, w, u, c]
max_check_attempts	# ^(*)
check_interval	# ^(*)
retry_interval	# ^(*)
active_checks_enabled	[0/1]
passive_checks_enabled	[0/1]
check_period	timeperiod_name ^(*)
obsess_over_service	[0/1]
check_freshness	[0/1]
freshness_threshold	#
event_handler	command_name
event_handler_enabled	[0/1]
low_flap_threshold	#
high_flap_threshold	#
flap_detection_enabled	[0/1]
flap_detection_options	[o, w, c, u]
process_perf_data	[0/1]
retain_status_information	[0/1]
retain_nonstatus_information	[0/1]
notification_interval	# ^(*)
first_notification_delay	#
notification_period	timeperiod_name ^(*)
notification_options	[w, u, c, r, f, s]
notifications_enabled	[0/1]
contacts	contacts ^(*)
contact_groups	contact_groups ^(*)
stalking_options	[o, w, u, c]
notes	note_string
notes_url	url
action_url	url
icon_image	image_file

```
        icon_image_alt    alt_string
    ...
}
```

定义样例:

```
define service{
    host_name            linux-server
    service_description  check-disk-sda1
    check_command        check-disk!/dev/sda1
    max_check_attempts   5
    check_interval       5
    retry_interval       3
    check_period         24x7
    notification_interval 30
    notification_period  24x7
    notification_options w,c,r
    contact_groups       linux-admins
}
```

域描述:

host_name: This directive is used to specify the **short name(s)** of the [host\(s\)](#) that the service “runs” on or is associated with. Multiple hosts should be separated by commas.

hostgroup_name: This directive is used to specify the **short name(s)** of the [hostgroup\(s\)](#) that the service “runs” on or is associated with. Multiple hostgroups should be separated by commas. The hostgroup_name may be used instead of, or in addition to, the host_name directive.

service_description;: This directive is used to define the description of the service, which may contain spaces, dashes, and colons (semicolons, apostrophes, and quotation marks should be avoided). No two services associated with the same host can have the same description. Services are uniquely identified with their **host_name** and **service_description** directives.

display_name: This directive is used to define an alternate name that should be displayed in the web interface for this service. If not specified, this defaults to the value you specify

for the **service_description** directive. Note: The current CGIs do not use this option, although future versions of the web interface will.

servicegroups: This directive is used to identify the **short name(s)** of the [servicegroup\(s\)](#) that the service belongs to. Multiple servicegroups should be separated by commas. This directive may be used as an alternative to using the **members** directive in [servicegroup definitions](#).

is_volatile: This directive is used to denote whether the service is "volatile". Services are normally **not** volatile. More information on volatile service and how they differ from normal services can be found [here](#). Value: 0 = service is not volatile, 1 = service is volatile.

check_command: This directive is used to specify the **short name** of the [command](#) that Nagios will run in order to check the status of the service. The maximum amount of time that the service check command can run is controlled by the [service check timeout](#) option.

initial_state: By default Nagios will assume that all services are in OK states when in starts. You can override the initial state for a service by using this directive. Valid options are: o = 正常(OK), w = 告警(WARNING), u = 未知(UNKNOWN), and c = 紧急(CRITICAL).

max_check_attempts: This directive is used to define the number of times that Nagios will retry the service check command if it returns any state other than an OK state. Setting this value to 1 will cause Nagios to generate an alert without retrying the service check again.

check_interval: This directive is used to define the number of "time units" to wait before scheduling the next "regular" check of the service. "Regular" checks are those that occur when the service is in an OK state or when the service is in a non-OK state, but has already been rechecked max_attempts number of times. Unless you've changed the [interval length](#) directive from the default value of 60, this number will mean minutes. More information on this value can be found in the [check scheduling](#) documentation.

retry_interval: This directive is used to define the number of "time units" to wait before scheduling a re-check of the service. Services are rescheduled at the retry interval when the have changed to a non-OK state. Once the service has been retried max_attempts times without a change in its status, it will revert to being scheduled at its "normal" rate as defined by the check_interval value. Unless you've changed the [interval length](#) directive from the default value of 60, this number will mean minutes. More information on this value can be found in the [check scheduling](#) documentation.

active_checks_enabled **: This directive is used to determine whether or not active checks of this service are enabled. Values: 0 = disable active service checks, 1 = enable active service checks.

passive_checks_enabled **: This directive is used to determine whether or not passive checks of this service are enabled. Values: 0 = disable passive service checks, 1 = enable passive service checks.

check_period: This directive is used to specify the short name of the [time period](#) during which active checks of this service can be made.

obsess_over_service **: This directive determines whether or not checks for the service will be "obsessed" over using the [ocsp_command](#).

check_freshness **: This directive is used to determine whether or not [freshness checks](#) are enabled for this service. Values: 0 = disable freshness checks, 1 = enable freshness checks.

freshness_threshold: This directive is used to specify the freshness threshold (in seconds) for this service. If you set this directive to a value of 0, Nagios will determine a freshness threshold to use automatically.

event_handler_enabled **: This directive is used to determine whether or not the event handler for this service is enabled. Values: 0 = disable service event handler, 1 = enable service event handler.

low_flap_threshold: This directive is used to specify the low state change threshold used in flap detection for this service. More information on flap detection can be found [here](#). If you set this directive to a value of 0, the program-wide value specified by the [low_service_flap_threshold](#) directive will be used.

high_flap_threshold: This directive is used to specify the high state change threshold used in flap detection for this service. More information on flap detection can be found [here](#). If you set this directive to a value of 0, the program-wide value specified by the [high_service_flap_threshold](#) directive will be used.

flap_detection_enabled **: This directive is used to determine whether or not flap detection is enabled for this service. More information on flap detection can be found [here](#). Values: 0 = disable service flap detection, 1 = enable service flap detection.

flap_detection_options: This directive is used to determine what service states the [flap detection logic](#) will use for this service. Valid options are a combination of one or more of the following: o = OK states, w = WARNING states, c = CRITICAL states, u = UNKNOWN states.

process_perf_data **: This directive is used to determine whether or not the processing of performance data is enabled for this service. Values: 0 = disable performance data processing, 1 = enable performance data processing.

retain_status_information: This directive is used to determine whether or not status-related information about the service is retained across program restarts. This is only useful if you have enabled state retention using the [retain state information](#) directive. Value: 0 = disable status information retention, 1 = enable status information retention.

retain_nonstatus_information: This directive is used to determine whether or not non-status information about the service is retained across program restarts. This is only useful if you have enabled state retention using the [retain state information](#) directive. Value: 0 = disable non-status information retention, 1 = enable non-status information retention.

notification_interval: This directive is used to define the number of "time units" to wait before re-notifying a contact that this service is **still** in a non-OK state. Unless you've changed the [interval length](#) directive from the default value of 60, this number will mean minutes. If you set this value to 0, Nagios will **not** re-notify contacts about problems for this service – only one problem notification will be sent out.

first_notification_delay: This directive is used to define the number of "time units" to wait before sending out the first problem notification when this service enters a non-OK state. Unless you've changed the [interval length](#) directive from the default value of 60, this number will mean minutes. If you set this value to 0, Nagios will start sending out notifications immediately.

notification_period: This directive is used to specify the short name of the [time period](#) during which notifications of events for this service can be sent out to contacts. No service notifications will be sent out during times which is not covered by the time period.

notification_options: This directive is used to determine when notifications for the service should be sent out. Valid options are a combination of one or more of the following: w = send notifications on a WARNING state, u = send notifications on an UNKNOWN state, c = send notifications on a CRITICAL state, r = send notifications on recoveries (OK state), f = send

notifications when the service starts and stops [flapping](#), and s = send notifications when [scheduled downtime](#) starts and ends. If you specify n (none) as an option, no service notifications will be sent out. If you do not specify any notification options, Nagios will assume that you want notifications to be sent out for all possible states. Example: If you specify w,r in this field, notifications will only be sent out when the service goes into a WARNING state and when it recovers from a WARNING state.

notifications_enabled [**](#): This directive is used to determine whether or not notifications for this service are enabled. Values: 0 = disable service notifications, 1 = enable service notifications.

contacts: This is a list of the **short names** of the [contacts](#) that should be notified whenever there are problems (or recoveries) with this service. Multiple contacts should be separated by commas. Useful if you want notifications to go to just a few people and don't want to configure [contact groups](#). You must specify at least one contact or contact group in each service definition.

contact_groups: This is a list of the **short names** of the [contact groups](#) that should be notified whenever there are problems (or recoveries) with this service. Multiple contact groups should be separated by commas. You must specify at least one contact or contact group in each service definition.

stalking_options: This directive determines which service states "stalking" is enabled for. Valid options are a combination of one or more of the following: o = stalk on OK states, w = stalk on WARNING states, u = stalk on UNKNOWN states, and c = stalk on CRITICAL states. More information on state stalking can be found [here](#).

notes: This directive is used to define an optional string of notes pertaining to the service. If you specify a note here, you will see it in the [extended information](#) CGI (when you are viewing information about the specified service).

notes_url: This directive is used to define an optional URL that can be used to provide more information about the service. If you specify an URL, you will see a red folder icon in the CGIs (when you are viewing service information) that links to the URL you specify here. Any valid URL can be used. If you plan on using relative paths, the base path will be the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`). This can be very useful if you want to make detailed information on the service, emergency contact methods, etc. available to other support staff.

action_url: This directive is used to define an optional URL that can be used to provide more actions to be performed on the service. If you specify an URL, you will see a red "splat" icon in the CGIs (when you are viewing service information) that links to the URL you specify here. Any valid URL can be used. If you plan on using relative paths, the base path will be the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`).

icon_image: This variable is used to define the name of a GIF, PNG, or JPG image that should be associated with this host. This image will be displayed in the [status](#) and [extended information](#) CGIs. The image will look best if it is 40x40 pixels in size. Images for hosts are assumed to be in the `logos/` subdirectory in your HTML images directory (i.e. `/usr/local/nagios/share/images/logos`).

icon_image_alt: This variable is used to define an optional string that is used in the ALT tag of the image specified by the `<icon_image>` argument. The ALT tag is used in the [status](#), [extended information](#) and [statusmap](#) CGIs.

6.1.8. 服务组定义

描述:

A service group definition is used to group one or more services together for simplifying configuration with [object tricks](#) or display purposes in the [CGIs](#).

定义格式:

注意

标记了^(*)的域是必备的而黑色是可选的。

```
define servicegroup{
    servicegroup_name      servicegroup_name(*)
    alias                  alias(*)
    members                services
    servicegroup_members   servicegroups
    notes                  note_string
    notes_url              url
    action_url             url
    ...
}
```

定义样例:

```
define servicegroup{
    servicegroup_name dbservices
    alias                Database Services
    members              ms1, SQL Server, ms1, SQL Server Agent, ms1, SQL DTC
}
```

域描述:

servicegroup_name: This directive is used to define a short name used to identify the service group.

alias: This directive is used to define is a longer name or description used to identify the service group. It is provided in order to allow you to more easily identify a particular service group.

members: This is a list of the **descriptions** of [service](#) (and the names of their corresponding hosts) that should be included in this group. Host and service names should be separated by commas. This directive may be used as an alternative to the **servicegroups** directive in [service definitions](#). The format of the member directive is as follows (note that a host name must precede a service name/description): members=<host1>,<service1>,<host2>,<service2>,...,<hostn>,<servicen>

servicegroup_members: This optional directive can be used to include services from other "sub" service groups in this service group. Specify a comma-delimited list of short names of other service groups whose members should be included in this group.

notes: This directive is used to define an optional string of notes pertaining to the service group. If you specify a note here, you will see the it in the [extended information](#) CGI (when you are viewing information about the specified service group).

notes_url: This directive is used to define an optional URL that can be used to provide more information about the service group. If you specify an URL, you will see a red folder icon in the CGIs (when you are viewing service group information) that links to the URL you specify here. Any valid URL can be used. If you plan on using relative paths, the base path will the the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`). This can be very useful if you want to make detailed information on the service group, emergency contact methods, etc. available to other support staff.

action_url: This directive is used to define an optional URL that can be used to provide more actions to be performed on the service group. If you specify an URL, you will see a red "splat" icon in the CGIs (when you are viewing service group information) that links to the URL you specify here. Any valid URL can be used. If you plan on using relative paths, the base path will be the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`).

6.1.9. 联系人定义

描述:

A contact definition is used to identify someone who should be contacted in the event of a problem on your network. The different arguments to a contact definition are described below.

定义格式:

注意

标记了^(*)的域是必备的而黑色是可选的。

```
define contact{
    contact_name          contact_name(*)
    alias                 alias(*)
    contactgroups         contactgroup_names
    host_notifications_enabled    [0/1](*)
    service_notifications_enabled [0/1](*)
    host_notification_period      timeperiod_name(*)
    service_notification_period   timeperiod_name(*)
    host_notification_options     [d,u,r,f,s,n](*)
    service_notification_options  [w,u,c,r,f,s,n](*)
    host_notification_commands    command_name(*)
    service_notification_commands command_name(*)
    email                    email_address
    pager                    pager_number or pager_email_gateway
    addressx                 additional_contact_address
    can_submit_commands      [0/1]
    retain_status_information [0/1]
    retain_nonstatus_information [0/1]
    ...
}
```

```
}
```

定义样例:

```
define contact{

    contact_name                jdoe

    alias                       John Doe

    host_notifications_enabled  1

    service_notifications_enabled  1

    service_notification_period 24x7

    host_notification_period    24x7

    service_notification_options w, u, c, r

    host_notification_options    d, u, r

    service_notification_commands notify-by-email

    host_notification_commands  host-notify-by-email

    email                       jdoe@localhost.localdomain

    pager                       555-5555@pagergateway.localhost.localdomain

    address1                    xxxxx.xyyy@icq.com

    address2                    555-555-5555

    can_submit_commands         1

}
```

域描述:

contact_name: This directive is used to define a short name used to identify the contact. It is referenced in [contact group definitions](#). Under the right circumstances, the `$CONTACTNAME$` [macro](#) will contain this value.

alias: This directive is used to define a longer name or description for the contact. Under the right circumstances, the `$CONTACTALIAS$` [macro](#) will contain this value.

contactgroups: This directive is used to identify the **short name(s)** of the [contactgroup\(s\)](#) that the contact belongs to. Multiple contactgroups should be separated by commas. This directive may be used as an alternative to (or in addition to) using the **members** directive in [contactgroup definitions](#).

host_notifications_enabled: This directive is used to determine whether or not the contact will receive notifications about host problems and recoveries. Values: 0 = don't send notifications, 1 = send notifications.

service_notifications_enabled: This directive is used to determine whether or not the contact will receive notifications about service problems and recoveries. Values: 0 = don't send notifications, 1 = send notifications.

host_notification_period: This directive is used to specify the short name of the [time period](#) during which the contact can be notified about host problems or recoveries. You can think of this as an "on call" time for host notifications for the contact. Read the documentation on [time periods](#) for more information on how this works and potential problems that may result from improper use.

service_notification_period: This directive is used to specify the short name of the [time period](#) during which the contact can be notified about service problems or recoveries. You can think of this as an "on call" time for service notifications for the contact. Read the documentation on [time periods](#) for more information on how this works and potential problems that may result from improper use.

host_notification_commands: This directive is used to define a list of the **short names** of the [commands](#) used to notify the contact of a **host** problem or recovery. Multiple notification commands should be separated by commas. All notification commands are executed when the contact needs to be notified. The maximum amount of time that a notification command can run is controlled by the [notification timeout](#) option.

host_notification_options: This directive is used to define the host states for which notifications can be sent out to this contact. Valid options are a combination of one or more of the following: d = notify on DOWN host states, u = notify on UNREACHABLE host states, r = notify on host recoveries (UP states), f = notify when the host starts and stops [flapping](#), and s = send notifications when host or service [scheduled downtime](#) starts and ends. If you specify n (none) as an option, the contact will not receive any type of host notifications.

service_notification_options: This directive is used to define the service states for which notifications can be sent out to this contact. Valid options are a combination of one or more of the following: w = notify on WARNING service states, u = notify on UNKNOWN service states, c = notify on CRITICAL service states, r = notify on service recoveries (OK states), and f =

notify when the service starts and stops [flapping](#). If you specify n (none) as an option, the contact will not receive any type of service notifications.

service_notification_commands: This directive is used to define a list of the **short names** of the [commands](#) used to notify the contact of a **service** problem or recovery. Multiple notification commands should be separated by commas. All notification commands are executed when the contact needs to be notified. The maximum amount of time that a notification command can run is controlled by the [notification timeout](#) option.

email: This directive is used to define an email address for the contact. Depending on how you configure your notification commands, it can be used to send out an alert email to the contact. Under the right circumstances, the \$CONTACTEMAIL\$ [macro](#) will contain this value.

pager: This directive is used to define a pager number for the contact. It can also be an email address to a pager gateway (i.e. pagejoe@pagenet.com). Depending on how you configure your notification commands, it can be used to send out an alert page to the contact. Under the right circumstances, the \$CONTACTPAGER\$ [macro](#) will contain this value.

addressx: Address directives are used to define additional "addresses" for the contact. These addresses can be anything – cell phone numbers, instant messaging addresses, etc. Depending on how you configure your notification commands, they can be used to send out an alert to the contact. Up to six addresses can be defined using these directives (**address1** through **address6**). The \$CONTACTADDRESSx\$ [macro](#) will contain this value.

can_submit_commands: This directive is used to determine whether or not the contact can submit [external commands](#) to Nagios from the CGIs. Values: 0 = don't allow contact to submit commands, 1 = allow contact to submit commands.

retain_status_information: This directive is used to determine whether or not status-related information about the contact is retained across program restarts. This is only useful if you have enabled state retention using the [retain state information](#) directive. Value: 0 = disable status information retention, 1 = enable status information retention.

retain_nonstatus_information: This directive is used to determine whether or not non-status information about the contact is retained across program restarts. This is only useful if you have enabled state retention using the [retain state information](#) directive. Value: 0 = disable non-status information retention, 1 = enable non-status information retention.

6.1.10. 联系人组定义

描述:

A contact group definition is used to group one or more [contacts](#) together for the purpose of sending out alert/recovery [notifications](#).

定义格式:

注意

标记了(*)的域是必备的而黑色是可选的。

```
define contactgroup{
    contactgroup_name      contactgroup_name(*)
    alias                  alias(*)
    members                contacts(*)
    contactgroup_members   contactgroups
    ...
}
```

定义样例:

```
define contactgroup{
    contactgroup_name      novell-admins
    alias                  Novell Administrators
    members                jdoe, rtobert, tzach
}
```

域描述:

contactgroup_name: This directive is a short name used to identify the contact group.

alias: This directive is used to define a longer name or description used to identify the contact group.

members: This directive is used to define a list of the **short names** of [contacts](#) that should be included in this group. Multiple contact names should be separated by commas. This directive may be used as an alternative to (or in addition to) using the **contactgroups** directive in [contact definitions](#).

contactgroup_members: This optional directive can be used to include contacts from other "sub" contact groups in this contact group. Specify a comma-delimited list of short names of other contact groups whose members should be included in this group.

6.1.11. 时间周期定义

描述：

A time period is a list of times during various days that are considered to be "valid" times for notifications and service checks. It consists of time ranges for each day of the week that "rotate" once the week has come to an end. Different types of exceptions to the normal weekly time are supported, including: specific weekdays, days of generic months, days of specific months, and calendar dates.

定义格式：

注意

标记了^(*)的域是必备的而黑色是可选的。

```
define timeperiod{
    timeperiod_name      timeperiod_name(*)
    alias                alias(*)
    [weekday]            timeranges
    [exception]          timeranges
    exclude              [timeperiod1,timeperiod2,...,timeperiodn]
    ...
}
```

定义样例：

```
define timeperiod{
    timeperiod_name      nonworkhours
    alias                Non-Work Hours
    sunday               00:00-24:00           ; Every Sunday
of every week
    monday               00:00-09:00, 17:00-24:00 ; Every Monday
of every week
    tuesday              00:00-09:00, 17:00-24:00 ; Every
Tuesday of every week
    wednesday            00:00-09:00, 17:00-24:00 ;
Every Wednesday of every week
}
```

```

        thursday                00:00-09:00, 17:00-24:00        ; Every
Thursday of every week

        friday                  00:00-09:00, 17:00-24:00        ; Every Friday
of every week

        saturday               00:00-24:00                      ; Every
Saturday of every week
    }

    define timeperiod{
        timeperiod_name        misc-single-days
        alias                   Misc Single Days
        1999-01-28              00:00-24:00                    ; January 28th, 1999
        monday 3                 00:00-24:00                    ; 3rd Monday of every
month
        day 2                   00:00-24:00                    ; 2nd day of every month
        february 10             00:00-24:00                    ; February 10th of every
year
        february -1             00:00-24:00                    ; Last day in February
of every year
        friday -2               00:00-24:00                    ; 2nd to last
Friday of every month
        thursday -1 november    00:00-24:00                    ; Last Thursday in
November of every year
    }

    define timeperiod{
        timeperiod_name        misc-date-ranges
        alias                   Misc Date Ranges
        2007-01-01 - 2008-02-01 00:00-24:00                    ; January 1st, 2007 to
February 1st, 2008

```

```

monday 3 - thursday 4      00:00-24:00      ; 3rd Monday to 4th
Thursday of every month

day 1 - 15                  00:00-24:00      ; 1st to 15th day of
every month

day 20 - -1                 00:00-24:00      ; 20th to the last day of
every month

july 10 - 15                00:00-24:00      ; July 10th to July 15th
of every year

april 10 - may 15            00:00-24:00      ; April 10th to May 15th
of every year

tuesday 1 april - friday 2 may  00:00-24:00      ; 1st Tuesday in April
to 2nd Friday in May of every year
}

define timeperiod{
    timeperiod_name      misc-skip-ranges
    alias                 Misc Skip Ranges
    2007-01-01 - 2008-02-01 / 3      00:00-24:00      ; Every 3 days
from January 1st, 2007 to February 1st, 2008
    2008-04-01 / 7      00:00-24:00      ; Every 7 days from
April 1st, 2008 (continuing forever)
    monday 3 - thursday 4 / 2      00:00-24:00      ; Every other day from
3rd Monday to 4th Thursday of every month
    day 1 - 15 / 5      00:00-24:00      ; Every 5 days from the
1st to the 15th day of every month
    july 10 - 15 / 2      00:00-24:00      ; Every other day from
July 10th to July 15th of every year
    tuesday 1 april - friday 2 may / 6 00:00-24:00      ; Every 6 days from the
1st Tuesday in April to the 2nd Friday in May of every year
}

```

域描述:

timeperiod_name: This directive is the short name used to identify the time period.

alias: This directive is a longer name or description used to identify the time period.

[weekday]: The weekday directives ("**sunday**" through "**saturday**") are comma-delimited lists of time ranges that are "valid" times for a particular day of the week. Notice that there are seven different days for which you can define time ranges (Sunday through Saturday). Each time range is in the form of HH:MM-HH:MM, where hours are specified on a 24 hour clock. For programlisting, 00:15-24:00 means 12:15am in the morning for this day until 12:20am midnight (a 23 hour, 45 minute total time range). If you wish to exclude an entire day from the timeperiod, simply do not include it in the timeperiod definition.

[exception]: You can specify several different types of exceptions to the standard rotating weekday schedule. Exceptions can take a number of different forms including single days of a specific or generic month, single weekdays in a month, or single calendar dates. You can also specify a range of days/dates and even specify skip intervals to obtain functionality described by "every 3 days between these dates". Rather than list all the possible formats for exception strings, I'll let you look at the programlisting timeperiod definitions above to see what's possible. :-) Weekdays and different types of exceptions all have different levels of precedence, so its important to understand how they can affect each other. More information on this can be found in the documentation on [timeperiods](#).

exclude: This directive is used to specify the short names of other timeperiod definitions whose time ranges should be excluded from this timeperiod. Multiple timeperiod names should be separated with a comma.

6.1.12. 命令定义

描述:

A command definition is just that. It defines a command. Commands that can be defined include service checks, service notifications, service event handlers, host checks, host notifications, and host event handlers. Command definitions can contain [macros](#), but you must make sure that you include only those macros that are "valid" for the circumstances when the command will be used. More information on what macros are available and when they are "valid" can be found [here](#). The different arguments to a command definition are outlined below.

定义格式:

注意

标记了(*)的域是必备的而黑色是可选的。

```
define command{  
    command_name      command_name(*)  
    command_line      command_line(*)  
    ...  
}
```

定义样例:

```
define command{  
    command_name      check_pop  
    command_line      /usr/local/nagios/libexec/check_pop -H $HOSTADDRESS$  
}
```

域描述:

command_name: This directive is the short name used to identify the command. It is referenced in [contact](#), [host](#), and [service definitions](#) (in notification, check, and event handler directives), among other places.

command_line: This directive is used to define what is actually executed by Nagios when the command is used for service or host checks, notifications, or [event handlers](#). Before the command line is executed, all valid [macros](#) are replaced with their respective values. See the documentation on macros for determining when you can use different macros. Note that the command line is **not** surrounded in quotes. Also, if you want to pass a dollar sign (\$) on the command line, you have to escape it with another dollar sign. **NOTE:** You may not include a semicolon (;) in the **command_line** directive, because everything after it will be ignored as a config file comment. You can work around this limitation by setting one of the [\\$USER\\$](#) macros in your [resource file](#) to a semicolon and then referencing the appropriate \$USER\$ macro in the **command_line** directive in place of the semicolon. you want to pass arguments to commands during runtime, you can use [\\$ARGn\\$](#) macros in the **command_line** directive of the command definition and then separate individual arguments from the command name (and from each other) using bang (!) characters in the object definition directive (host check command, service event handler command, etc) that references

the command. More information on how arguments in command definitions are processed during runtime can be found in the documentation on [macros](#).

6.1.13. 服务依赖定义

描述:

Service dependencies are an advanced feature of Nagios that allow you to suppress notifications and active checks of services based on the status of one or more other services. Service dependencies are optional and are mainly targeted at advanced users who have complicated monitoring setups. More information on how service dependencies work (read this!) can be found [here](#).

定义格式:

注意

标记了^(*)的域是必备的而黑色是可选的。然而你最少要在定义中给定出一种使用类型标准。

```
define servicedependency{  
    dependent_host_name      host_name(*)  
    dependent_hostgroup_name hostgroup_name  
    dependent_service_description service_description(*)  
    host_name                host_name(*)  
    hostgroup_name           hostgroup_name  
    service_description       service_description(*)  
    inherits_parent          [0/1]  
    execution_failure_criteria [o, w, u, c, p, n]  
    notification_failure_criteria [o, w, u, c, p, n]  
    dependency_period         timeperiod_name  
    ...  
}
```

定义样例:

```
define servicedependency{  
    host_name                WWW1  
    service_description       Apache Web Server  
    dependent_host_name       WWW1
```

```

dependent_service_description    Main Web Site
execution_failure_criterion
notification_failure_criteria    w, u, c
}

```

域描述:

dependent_host: This directive is used to identify the **short name(s)** of the [host\(s\)](#) that the **dependent** service “runs” on or is associated with. Multiple hosts should be separated by commas. Leaving this directive blank can be used to create [“same host” dependencies](#).

dependent_hostgroup: This directive is used to specify the **short name(s)** of the [hostgroup\(s\)](#) that the **dependent** service “runs” on or is associated with. Multiple hostgroups should be separated by commas. The `dependent_hostgroup` may be used instead of, or in addition to, the `dependent_host` directive.

dependent_service_description: This directive is used to identify the **description** of the **dependent**[service](#).

host_name: This directive is used to identify the **short name(s)** of the [host\(s\)](#) that the service **that is being depended upon** (also referred to as the master service) “runs” on or is associated with. Multiple hosts should be separated by commas.

hostgroup_name: This directive is used to identify the **short name(s)** of the [hostgroup\(s\)](#) that the service **that is being depended upon** (also referred to as the master service) “runs” on or is associated with. Multiple hostgroups should be separated by commas. The `hostgroup_name` may be used instead of, or in addition to, the `host_name` directive.

service_description: This directive is used to identify the **description** of the [service](#) **that is being depended upon** (also referred to as the master service).

inherits_parent: This directive indicates whether or not the dependency inherits dependencies of the service **that is being depended upon** (also referred to as the master service). In other words, if the master service is dependent upon other services and any one of those dependencies fail, this dependency will also fail.

execution_failure_criteria: This directive is used to specify the criteria that determine when the dependent service should **not** be actively checked. If the **master** service is in one of the failure states we specify, the **dependent** service will not be actively checked. Valid options

are a combination of one or more of the following (multiple options are separated with commas):
o = fail on an OK state, w = fail on a WARNING state, u = fail on an UNKNOWN state, c = fail on a CRITICAL state, and p = fail on a pending state (e.g. the service has not yet been checked).
If you specify n (none) as an option, the execution dependency will never fail and checks of the dependent service will always be actively checked (if other conditions allow for it to be).
Example: If you specify o,c,u in this field, the **dependent** service will not be actively checked if the **master** service is in either an OK, a CRITICAL, or an UNKNOWN state.

notification_failure_criteria: This directive is used to define the criteria that determine when notifications for the dependent service should **not** be sent out. If the **master** service is in one of the failure states we specify, notifications for the **dependent** service will not be sent to contacts. Valid options are a combination of one or more of the following: o = fail on an OK state, w = fail on a WARNING state, u = fail on an UNKNOWN state, c = fail on a CRITICAL state, and p = fail on a pending state (e.g. the service has not yet been checked). If you specify n (none) as an option, the notification dependency will never fail and notifications for the dependent service will always be sent out. Example: If you specify w in this field, the notifications for the **dependent** service will not be sent out if the **master** service is in a WARNING state.

dependency_period: This directive is used to specify the short name of the [time period](#) during which this dependency is valid. If this directive is not specified, the dependency is considered to be valid during all times.

6.1.14. 服务扩展定义

描述:

Service escalations are **completely optional** and are used to escalate notifications for a particular service. More information on how notification escalations work can be found [here](#).

定义格式:

注意

标记了(*)的域是必备的而黑色是可选的。

```
define serviceescalation{  
    host_name          host_name(*)  
    hostgroup_name      hostgroup_name  
    service_description service_description(*)  
}
```

```

contacts          contacts(*)
contact_groups    contactgroup_name(*)
first_notification #(*)
last_notification #(*)
notification_interval #(*)
escalation_period timeperiod_name
escalation_options [w, u, c, r]
...
}

```

定义样例:

```

define serviceescalation{
    host_name          nt-3
    service_description Processor Load
    first_notification  4
    last_notification 0
    notification_interval 30
    contact_groups      all-nt-admins, themanagers
}

```

域描述:

host_name: This directive is used to identify the **short name(s)** of the [host\(s\)](#) that the [service escalation](#) should apply to or is associated with.

hostgroup_name: This directive is used to specify the **short name(s)** of the [hostgroup\(s\)](#) that the service escalation should apply to or is associated with. Multiple hostgroups should be separated by commas. The hostgroup_name may be used instead of, or in addition to, the host_name directive.

service_description: This directive is used to identify the **description** of the [service](#) the escalation should apply to.

first_notification: This directive is a number that identifies the **first** notification for which this escalation is effective. For instance, if you set this value to 3, this escalation

will only be used if the service is in a non-OK state long enough for a third notification to go out.

last_notification: This directive is a number that identifies the **last** notification for which this escalation is effective. For instance, if you set this value to 5, this escalation will not be used if more than five notifications are sent out for the service. Setting this value to 0 means to keep using this escalation entry forever (no matter how many notifications go out).

contacts: This is a list of the **short names** of the [contacts](#) that should be notified whenever there are problems (or recoveries) with this service. Multiple contacts should be separated by commas. Useful if you want notifications to go to just a few people and don't want to configure [contact groups](#). You must specify at least one contact or contact group in each service escalation definition.

contact_groups: This directive is used to identify the **short name** of the [contact group](#) that should be notified when the service notification is escalated. Multiple contact groups should be separated by commas. You must specify at least one contact or contact group in each service escalation definition.

notification_interval: This directive is used to determine the interval at which notifications should be made while this escalation is valid. If you specify a value of 0 for the interval, Nagios will send the first notification when this escalation definition is valid, but will then prevent any more problem notifications from being sent out for the host. Notifications are sent out again until the host recovers. This is useful if you want to stop having notifications sent out after a certain amount of time. Note: If multiple escalation entries for a host overlap for one or more notification ranges, the smallest notification interval from all escalation entries is used.

escalation_period: This directive is used to specify the short name of the [time period](#) during which this escalation is valid. If this directive is not specified, the escalation is considered to be valid during all times.

escalation_options: This directive is used to define the criteria that determine when this service escalation is used. The escalation is used only if the service is in one of the states specified in this directive. If this directive is not specified in a service escalation, the escalation is considered to be valid during all service states. Valid options are a combination of one or more of the following: r = escalate on an OK (recovery) state, w = escalate on a WARNING

state, u = escalate on an UNKNOWN state, and c = escalate on a CRITICAL state. Example: If you specify w in this field, the escalation will only be used if the service is in a WARNING state.

6.1.15. 主机依赖定义

描述:

Host dependencies are an advanced feature of Nagios that allow you to suppress notifications for hosts based on the status of one or more other hosts. Host dependencies are optional and are mainly targeted at advanced users who have complicated monitoring setups. More information on how host dependencies work (read this!) can be found [here](#).

定义格式:

注意

标记了(*)的域是必备的黑体是可选的。

```
define hostdependency{
    dependent_host_name      host_name(*)
    dependent_hostgroup_name  hostgroup_name
    host_name                 host_name(*)
    hostgroup_name            hostgroup_name
    inherits_parent           [0/1]
    execution_failure_criteria [o, d, u, p, n]
    notification_failure_criteria [o, d, u, p, n]
    dependency_period         timeperiod_name
    ...
}
```

定义样例:

```
define hostdependency{
    host_name                WWW1
    dependent_host_name       DBASE1
    notification_failure_criteria d, u
}
```

域描述:

dependent_host_name: This directive is used to identify the **short name(s)** of the **dependent**[host\(s\)](#). Multiple hosts should be separated by commas.

dependent_hostgroup_name: This directive is used to identify the **short name(s)** of the **dependent**[hostgroup\(s\)](#). Multiple hostgroups should be separated by commas. The `dependent_hostgroup_name` may be used instead of, or in addition to, the `dependent_host_name` directive.

host_name: This directive is used to identify the **short name(s)** of the [host\(s\)](#) **that is being depended upon** (also referred to as the master host). Multiple hosts should be separated by commas.

hostgroup_name: This directive is used to identify the **short name(s)** of the [hostgroup\(s\)](#) **that is being depended upon** (also referred to as the master host). Multiple hostgroups should be separated by commas. The `hostgroup_name` may be used instead of, or in addition to, the `host_name` directive.

inherits_parent: This directive indicates whether or not the dependency inherits dependencies of the host **that is being depended upon** (also referred to as the master host). In other words, if the master host is dependent upon other hosts and any one of those dependencies fail, this dependency will also fail.

execution_failure_criteria: This directive is used to specify the criteria that determine when the dependent host should **not** be actively checked. If the **master** host is in one of the failure states we specify, the **dependent** host will not be actively checked. Valid options are a combination of one or more of the following (multiple options are separated with commas): o = fail on an UP state, d = fail on a DOWN state, u = fail on an UNREACHABLE state, and p = fail on a pending state (e.g. the host has not yet been checked). If you specify n (none) as an option, the execution dependency will never fail and the dependent host will always be actively checked (if other conditions allow for it to be). Example: If you specify u,d in this field, the **dependent** host will not be actively checked if the **master** host is in either an UNREACHABLE or DOWN state.

notification_failure_criteria: This directive is used to define the criteria that determine when notifications for the dependent host should **not** be sent out. If the **master** host is in one of the failure states we specify, notifications for the **dependent** host will not be sent to contacts. Valid options are a combination of one or more of the following: o = fail on an UP state, d = fail on a DOWN state, u = fail on an UNREACHABLE state, and p = fail on a pending state (e.g. the host has not yet been checked). If you specify n (none) as an option, the notification

dependency will never fail and notifications for the dependent host will always be sent out. Example: If you specify `d` in this field, the notifications for the **dependent** host will not be sent out if the **master** host is in a DOWN state.

dependency_period: This directive is used to specify the short name of the [time period](#) during which this dependency is valid. If this directive is not specified, the dependency is considered to be valid during all times.

6.1.16. 主机扩展定义

描述:

Host escalations are **completely optional** and are used to escalate notifications for a particular host. More information on how notification escalations work can be found [here](#).

定义格式:

注意

标记了^(*)的域是必备的而黑色是可选的。

```
define hostescalation{
    host_name          host_name(*)
    hostgroup_name      hostgroup_name
    contacts            contacts(*)
    contact_groups      contactgroup_name(*)
    first_notification   #(*)
    last_notification    #(*)
    notification_interval #(*)
    escalation_period    timeperiod_name
    escalation_options    [d, u, r]
    ...
}
```

定义样例:

```
define hostescalation{
    host_name          router-34
    first_notification    5
    last_notification 8
}
```



```
notification_interval    60
contact_groups           all-router-admins
}
```

域描述:

host_name: This directive is used to identify the **short name** of the [host](#) that the escalation should apply to.

hostgroup_name: This directive is used to identify the **short name(s)** of the [hostgroup\(s\)](#) that the escalation should apply to. Multiple hostgroups should be separated by commas. If this is used, the escalation will apply to all hosts that are members of the specified hostgroup(s).

first_notification: This directive is a number that identifies the **first** notification for which this escalation is effective. For instance, if you set this value to 3, this escalation will only be used if the host is down or unreachable long enough for a third notification to go out.

last_notification: This directive is a number that identifies the **last** notification for which this escalation is effective. For instance, if you set this value to 5, this escalation will not be used if more than five notifications are sent out for the host. Setting this value to 0 means to keep using this escalation entry forever (no matter how many notifications go out).

contacts: This is a list of the **short names** of the [contacts](#) that should be notified whenever there are problems (or recoveries) with this host. Multiple contacts should be separated by commas. Useful if you want notifications to go to just a few people and don't want to configure [contact groups](#). You must specify at least one contact or contact group in each host escalation definition.

contact_groups: This directive is used to identify the **short name** of the [contact group](#) that should be notified when the host notification is escalated. Multiple contact groups should be separated by commas. You must specify at least one contact or contact group in each host escalation definition.

notification_interval: This directive is used to determine the interval at which notifications should be made while this escalation is valid. If you specify a value of 0 for the interval, Nagios will send the first notification when this escalation definition is valid, but will then prevent any more problem notifications from being sent out for the host. Notifications are sent out again until the host recovers. This is useful if you want to stop

having notifications sent out after a certain amount of time. Note: If multiple escalation entries for a host overlap for one or more notification ranges, the smallest notification interval from all escalation entries is used.

escalation_period: This directive is used to specify the short name of the [time period](#) during which this escalation is valid. If this directive is not specified, the escalation is considered to be valid during all times.

escalation_options: This directive is used to define the criteria that determine when this host escalation is used. The escalation is used only if the host is in one of the states specified in this directive. If this directive is not specified in a host escalation, the escalation is considered to be valid during all host states. Valid options are a combination of one or more of the following: r = escalate on an UP (recovery) state, d = escalate on a DOWN state, and u = escalate on an UNREACHABLE state. Example: If you specify d in this field, the escalation will only be used if the host is in a DOWN state.

6.1.17. 额外主机信息定义

描述:

Extended host information entries are basically used to make the output from the [status](#), [statusmap](#), [statusurl](#), and [extinfo](#) CGIs look pretty. They have no effect on monitoring and are completely optional.



Tip: As of Nagios 3.x, all directives contained in extended host information definitions are also available in [host definitions](#). Thus, you can choose to define the directives below in your host definitions if it makes your configuration simpler. Separate extended host information definitions will continue to be supported for backward compatability.

定义格式:

注意

标记了(*)的域是必备的而黑色是可选的。然而你在定义里至少要提供一种可选域以使其有用。

```
define hostextinfo{
    host_name      host_name(*)
    notes          note_string
    notes_url      url
}
```

```

        action_url      url
        icon_image      image_file
        icon_image_alt   alt_string
        vrml_image       image_file
        statusmap_image  image_file
        2d_coords        x_coord, y_coord
        3d_coords        x_coord, y_coord, z_coord
        ...
    }

```

定义样例:

```

define hostextinfo{
    host_name      netware1
    notes          This is the primary Netware file server
    notes_url
    http://webserver.localhost.localdomain/hostinfo.pl?host=netware1
    icon_image     novell40.png
    icon_image_alt  IntranetWare 4.11
    vrml_image     novell40.png
    statusmap_image novell40.gd2
    2d_coords      100, 250
    3d_coords      100.0, 50.0, 75.0
}

```

Variable Descriptions:

host_name: This variable is used to identify the **short name** of the [host](#) which the data is associated with.

notes: This directive is used to define an optional string of notes pertaining to the host. If you specify a note here, you will see the it in the [extended information](#) CGI (when you are viewing information about the specified host).

notes_url: This variable is used to define an optional URL that can be used to provide more information about the host. If you specify an URL, you will see a link that says "Extra Host

Notes” in the [extended information](#) CGI (when you are viewing information about the specified host). Any valid URL can be used. If you plan on using relative paths, the base path will be the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`). This can be very useful if you want to make detailed information on the host, emergency contact methods, etc. available to other support staff.

action_url: This directive is used to define an optional URL that can be used to provide more actions to be performed on the host. If you specify an URL, you will see a link that says “Extra Host Actions” in the [extended information](#) CGI (when you are viewing information about the specified host). Any valid URL can be used. If you plan on using relative paths, the base path will be the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`).

icon_image: This variable is used to define the name of a GIF, PNG, or JPG image that should be associated with this host. This image will be displayed in the [status](#) and [extended information](#) CGIs. The image will look best if it is 40x40 pixels in size. Images for hosts are assumed to be in the `logos/` subdirectory in your HTML images directory (i.e. `/usr/local/nagios/share/images/logos`).

icon_image_alt: This variable is used to define an optional string that is used in the ALT tag of the image specified by the `<icon_image>` argument. The ALT tag is used in the [status](#), [extended information](#) and [statusmap](#) CGIs.

vrml_image: This variable is used to define the name of a GIF, PNG, or JPG image that should be associated with this host. This image will be used as the texture map for the specified host in the [statuswrl](#) CGI. Unlike the image you use for the `<icon_image>` variable, this one should probably **not** have any transparency. If it does, the host object will look a bit wierd. Images for hosts are assumed to be in the `logos/` subdirectory in your HTML images directory (i.e. `/usr/local/nagios/share/images/logos`).

statusmap_image: This variable is used to define the name of an image that should be associated with this host in the [statusmap](#) CGI. You can specify a JPEG, PNG, and GIF image if you want, although I would strongly suggest using a GD2 format image, as other image formats will result in a lot of wasted CPU time when the statusmap image is generated. GD2 images can be created from PNG images by using the `pngtogd2` utility supplied with Thomas Boutell’s [gd library](#). The GD2 images should be created in **uncompressed** format in order to minimize CPU load when the statusmap CGI is generating the network map image. The image will look best if it is 40x40 pixels

in size. You can leave these option blank if you are not using the statusmap CGI. Images for hosts are assumed to be in the logos/ subdirectory in your HTML images directory (i.e. `/usr/local/nagios/share/images/logos`).

2d_coords: This variable is used to define coordinates to use when drawing the host in the [statusmap](#) CGI. Coordinates should be given in positive integers, as the correspond to physical pixels in the generated image. The origin for drawing (0,0) is in the upper left hand corner of the image and extends in the positive x direction (to the right) along the top of the image and in the positive y direction (down) along the left hand side of the image. For reference, the size of the icons drawn is usually about 40x40 pixels (text takes a little extra space). The coordinates you specify here are for the upper left hand corner of the host icon that is drawn. Note: Don't worry about what the maximum x and y coordinates that you can use are. The CGI will automatically calculate the maximum dimensions of the image it creates based on the largest x and y coordinates you specify.

3d_coords: This variable is used to define coordinates to use when drawing the host in the [statuswrl](#) CGI. Coordinates can be positive or negative real numbers. The origin for drawing is (0.0, 0.0, 0.0). For reference, the size of the host cubes drawn is 0.5 units on each side (text takes a little more space). The coordinates you specify here are used as the center of the host cube.

6.1.18. 额外服务信息定义

描述:

Extended service information entries are basically used to make the output from the [status](#) and [extinfo](#) CGIs look pretty. They have no effect on monitoring and are completely optional.



Tip: As of Nagios 3.x, all directives contained in extended service information definitions are also available in [service definitions](#). Thus, you can choose to define the directives below in your service definitions if it makes your configuration simpler. Separate extended service information definitions will continue to be supported for backward compatability.

定义格式:

注意

标记了(*)的域是必备的而黑色是可选的。然而你在定义里至少要提供一个可选域以使其有用。

```

define serviceextinfo{
    host_name          host_name(*)
    service_description service_description(*)
    notes              note_string
    notes_url          url
    action_url         url
    icon_image         image_file
    icon_image_alt     alt_string
    ...
}

```

定义样例:

```

define serviceextinfo{
    host_name          linux2
    service_description Log Anomalies
    notes              Security-related log anomalies on secondary Linux
server
    notes_url
    http://webserver.localhost.localdomain/serviceinfo.pl?host=linux2&service=Log+Anoma
lies
    icon_image         security.png
    icon_image_alt     Security-Related Alerts
}

```

Variable Descriptions:

host_name: This directive is used to identify the **short name** of the host that the [service](#) is associated with.

service_description: This directive is description of the [service](#) which the data is associated with.

notes: This directive is used to define an optional string of notes pertaining to the service. If you specify a note here, you will see the it in the [extended information](#) CGI (when you are viewing information about the specified service).

notes_url: This directive is used to define an optional URL that can be used to provide more information about the service. If you specify an URL, you will see a link that says “Extra Service Notes” in the [extended information](#) CGI (when you are viewing information about the specified service). Any valid URL can be used. If you plan on using relative paths, the base path will be the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`). This can be very useful if you want to make detailed information on the service, emergency contact methods, etc. available to other support staff.

action_url: This directive is used to define an optional URL that can be used to provide more actions to be performed on the service. If you specify an URL, you will see a link that says “Extra Service Actions” in the [extended information](#) CGI (when you are viewing information about the specified service). Any valid URL can be used. If you plan on using relative paths, the base path will be the same as what is used to access the CGIs (i.e. `/cgi-bin/nagios/`).

icon_image: This variable is used to define the name of a GIF, PNG, or JPG image that should be associated with this host. This image will be displayed in the [status](#) and [extended information](#) CGIs. The image will look best if it is 40x40 pixels in size. Images for hosts are assumed to be in the `logos/` subdirectory in your HTML images directory (i.e. `/usr/local/nagios/share/images/logos`).

icon_image_alt: This variable is used to define an optional string that is used in the ALT tag of the image specified by the `<icon_image>` argument. The ALT tag is used in the [status](#), [extended information](#) and [statusmap](#) CGIs.

6.2. 对象定义的省时诀窍

或者是... “如何来让你保持清醒”

6.2.1. 介绍

本文试图向你解释如何让你利用那些隐藏于 [基于模板的对象定义](#) 之后的东西。那么你要问怎么来干？几各对象定义可以让你指定多个主机名和主机组名，允许你“复制”主机或服务的对象定义。我将逐个地说明支持这种方式的每种对象。如下的这些对象支持所要的省时特性：

- [服务](#)
- [服务扩展](#)
- [服务依赖](#)
- [主机扩展](#)
- [主机依赖](#)

- [主机组](#)

没有列出的对象类型（象时间范围、命令等）不支持以上特性我将作出说明。

6.2.2. 正则式匹配

下例中我将使用“标准”的对象名匹配式。如果你愿意，可以打开 [use_regexp_matching](#)配置选项里的使能开关。默认情况下只是对象名里包含*, ?, +或|..的作为正则式进行处理，如果你想让全部都认为是正则式，你应使能 [use_true_regexp_matching](#)配置选项。正则式可以被用于如下例子中的对象内的域（主机名称、主机组名、服务名称和服务组名）。

注意



使用正则时一定要小心—你可能需要修改配置文件，有时一些指令你并不想真正地被理解为正则式只是看起来角，任何问题都变成了你应验证你配置文件的证明。

6.2.3. 服务的定义

多个主机：如果你想在多个主机上创建同一个 [服务](#)，你可以在多个主机的**host_name**定义中实现。如下的定义中将服务名称叫**SOMESERVICE**的绑定在主机名字叫**HOST1**到**HOSTN**的多个主机上。所有的名字叫**SOMESERVICE**的服务将是同一个（例如有同一个检测命令、最大检测次数、告警周期等）。

```
define service{
    host_name          HOST1, HOST2, HOST3, ..., HOSTN
    service_description SOMESERVICE
    other service directives
    ...
}
```

在多个主机组里的全部主机：如果你想将一个或多个主机组里的全部主机标定同一个服务，该怎么办？在服务定义里的主机组域 **hostgroup_name** 里指定一个或多个玉机组。下面的服务名叫 **SOMESERVICE** 的服务被指定在一系列主机组 **HOSTGROUP1** 到 **HOSTGROUPN**。全部的名叫 **SOMESERVICE** 的服务将是同一个（例如有同样的检测命令、最大检测次数、告警周期等）。

```
define service{
    hostgroup_name      HOSTGROUP1, HOSTGROUP2, ..., HOSTGROUPN
    service_description SOMESERVICE
    other service directives
    ...
}
```



```
}
```

全部主机: 如果你想对你配置文件里的全部主机指定同一个服务, 你要在 **host_name** 域里使用通配符。下面将在配置文件里指定一个服务名叫 **SOMESERVICE** 的服务。全部的名叫 **SOMESERVICE** 的服务将是同一个 (例如相同的检测命令、最大检测次数、告警周期等)。

```
define service{
    host_name          *
    service_description SOMESERVICE
    other service directives
    ...
}
```

不包含主机: 如果你想定义一个服务在许多个主机或主机上但不包含某几个主机时, 可以在不包含的主机或主机组前加上 **!** 符号。

```
define service{
    host_name          HOST1, HOST2, !HOST3, !HOST4, ..., HOSTN
    hostgroup_name
HOSTGROUP1, HOSTGROUP2, !HOSTGROUP3, !HOSTGROUP4, ..., HOSTGROUPN
    service_description SOMESERVICE
    other service directives
    ...
}
```

6.2.4. 服务扩展的定义

多个主机: 如果想对多个主机上的服务或服务描述创建同一个 [服务扩展](#) 对象, 你可以在多个主机上指定 **host_name** 域。如下在主机系列从 **HOST1** 到 **HOSTN** 上指定一个服务扩展对象到服务名为 **SOMESERVICE** 的服务, 这些服务扩展将有同一个内容定义 (如相同的联系人组、通知间隔等)。

```
define serviceescalation{
    host_name          HOST1, HOST2, HOST3, ..., HOSTN
    service_description SOMESERVICE
    other escalation directives
    ...
}
```

```
}
```

多个主机里的全部主机：如果想对一个或多个主机组里的全部主机上的服务定义同一个服务扩展，你可以使用 **hostgroup_name** 域。下面将在主机组系列从 **HOSTGROUP1** 到 **HOSTGROUPN** 上全部主机上的服务名是 **SOMESERVICE** 有同一个服务扩展。所有的服务扩展是同一的(如有相同的联系人组、通知间隔)。

```
define serviceescalation{  
    hostgroup_name          HOSTGROUP1,HOSTGROUP2,...,HOSTGROUPN  
    service_description      SOMESERVICE  
    other_escalation_directives  
    ...  
}
```

全部主机：如果你想在你的配置文件里的全部主机上相同名称或描述的服务上创建同一个服务扩展，你需要在 **host_name** 域里用通配符。下面在配置文件里的全部主机上定义一个名为 **SOMESERVICE** 的服务有相同的服务扩展。全部的服务扩展是同一个(如有相同的联系人组、通知间隔等)。

```
define serviceescalation{  
    host_name                *  
    service_description      SOMESERVICE  
    other_escalation_directives  
    ...  
}
```

不包含主机：如果你想定义一个服务扩展在许多个主机或主机但不包含某几个主机上的服务时，可以在不包含>的主机或主机组前加上 **!**符号。

```
define serviceescalation{  
    host_name                HOST1,HOST2,!HOST3,!HOST4,...,HOSTN  
    hostgroup_name  
HOSTGROUP1,HOSTGROUP2,!HOSTGROUP3,!HOSTGROUP4,...,HOSTGROUPN  
    service_description      SOMESERVICE  
    other_escalation_directives  
    ...  
}
```

一个主机上的全部服务：如果想对某个特别的主机上全部的服务创建同一个 [服务扩展](#)，你可以在 **service_description** 域里使用通配符。下面在主机名是 **HOST1** 上的全部服务创建同一个服务扩展。如下的服务扩展将是同一个(如有相同的联系人组、通知间隔等)。

如果你特别喜欢急功冒进的话，你可以在 **host_name** 和 **service_description** 两个域里同时使用通配符。这样做将会创建一个你配置文件里的全部主机上的全部服务中定义同一个服务扩展。

```
define serviceescalation{  
    host_name            HOST1  
    service_description   *  
    other_escalation_directives  
    ...  
}
```

同一个主机上的多个服务：如果对某个主机上的一个或多个服务创建同一个 [服务扩展](#)，你可以在 **service_description** 域里指定服务描述。如下例中，在一主机名为 **HOST1** 上的一系列多个服务从 **SERVICE1** 到 **SERVICEN** 上创建服务扩展。所有的服务扩展是同一个(如有相同的联系人组、通知间隔等)。

```
define serviceescalation{  
    host_name            HOST1  
    service_description   SERVICE1, SERVICE2, ..., SERVICEN  
    other_escalation_directives  
    ...  
}
```

多个服务组里的全部服务：如果你想在一個或多个服务组里的全部服务创建同一个服务扩展，你可以用 **servicegroup_name** 域。如下将在一系列服务组自 **SERVICEGROUP1** 到 **SERVICEGROUPN** 的全部服务创建同一个服务扩展。这些服务扩展是同一个(如有相同的联系人组、通知间隔等)。

```
define serviceescalation{  
    servicegroup_name     SERVICEGROUP1, SERVICEGROUP2, ..., SERVICEGROUPN  
    other_escalation_directives  
    ...  
}
```

6.2.5. 服务依赖的定义

多个主机：如果想在多个主机上创建同名或相同描述的 [服务依赖](#)，你可以在多个主机定义里指定 `host_name` 或 `dependent_host_name` 域或是两者之一。在下例中，在主机 `HOST3` 和 `HOST4` 上的服务 `SERVICE2` 依赖于在 `HOST1` 和 `HOST2` 主机上的 `SERVICE1` 服务。所有的主机服务依赖定义是相同的，除了主机名称（如有相同的通知故障处理等）。

```
define servicedependency{  
    host_name                HOST1, HOST2  
    service_description      SERVICE1  
    dependent_host_name      HOST3, HOST4  
    dependent_service_description  SERVICE2  
    other dependency directives  
    ...  
}
```

多个主机组里的全部主机：如果你想在一个或多个主机组里的全部主机上创建一个同名或同描述的服务依赖，你可以指定 `hostgroup_name` 和 `dependent_hostgroup_name` 域或是两者之一。在下例中，主机组 `HOSTGROUP3` 和 `HOSTGROUP4` 里的全部主机上的服务 `SERVICE2` 将依赖于主机组 `HOSTGROUP1` 和 `HOSTGROUP2` 上的 `SERVICE1` 服务。假定每个主机组里有 5 个主机，那么这个定义将相当于创建了 100 个服务依赖！所有的服务依赖是相同的除了那些主机名有所不同（如有相同的通知故障处理等）。

```
define servicedependency{  
    hostgroup_name            HOSTGROUP1, HOSTGROUP2  
    service_description        SERVICE1  
    dependent_hostgroup_name  HOSTGROUP3, HOSTGROUP4  
    dependent_service_description  SERVICE2  
    other dependency directives  
    ...  
}
```

一个主机上的全部服务：如果你想创建针对某个主机的全部服务上的服务依赖，你可以在 `service_description` 和 `dependent_service_description` 域里使用通配符或是两者之一中使用。在下例中，全部在主机 `HOST2` 上的服务依赖于主机 `HOST1` 上的全部服务。全部的服务依赖将是相同的（如有相同的通知故障处理等）。

```
define servicedependency{
```

```

        host_name                HOST1
        service_description      *
        dependent_host_name      HOST2
        dependent_service_description  *
        other dependency directives
    ...
}

```

一个主机上的多个服务：如果你想创建对某个主机上的多个服务的服务依赖，你可以在 **service_description** 和 **dependent_service_description** 域里写一个或多个服务描述，象这样：

```

define servicedependency{
    host_name                HOST1
    service_description      SERVICE1, SERVICE2, ..., SERVICEN
    dependent_host_name      HOST2
    dependent_service_description  SERVICE1, SERVICE2, ..., SERVICEN
    other dependency directives
    ...
}

```

多个服务组里的全部服务：如果你想在一個或多个服务组里的全部服务上创建服务依赖，你可以用 **servicegroup_name** 和 **dependent_servicegroup_name** 域，象这样：

```

define servicedependency{
    servicegroup_name        SERVICEGROUP1, SERVICEGROUP2, ..., SERVICEGROUPN
    dependent_servicegroup_name
    SERVICEGROUP3, SERVICEGROUP4, ... SERVICEGROUPN
    other dependency directives
    ...
}

```

相同主机的服务依赖：如果想在相同主机的服务上创建服务依赖，空着 **dependent_host_name** 和 **dependent_hostgroup_name** 域。如下的例子中，主机 **HOST1** 和 **HOST2** 至少有四个服务绑定其上：**SERVICE1**、**SERVICE2**、**SERVICE3** 和 **SERVICE4**，在这个例子中，主机 **HOST1** 的 **SERVICE3** 和 **SERVICE4** 依赖于自身的

SERVICE1 和 SERVICE2 服务，相似的，HOST2 主机上 SERVICE3 和 SERVICE4 服务依赖于自身的 SERVICE1 和 SERVICE2 服务。

```
define servicedependency{  
    host_name                HOST1, HOST2  
    service_description      SERVICE1, SERVICE2  
    dependent_service_description  SERVICE3, SERVICE4  
    other dependency directives  
    ...  
}
```

6.2.6. 主机扩展的定义

多个主机：如果你想对多个主机创建同一个 [主机扩展](#)，你需要使用 **host_name** 域。如下将在一系列自 HOST1 到 HOSTN 的主机上创建同一的主机扩展。如下的主机扩展是同一个(如相同的联系人组、通知间隔等)。

```
define hostescalation{  
    host_name                HOST1, HOST2, HOST3, ..., HOSTN  
    other escalation directives  
    ...  
}
```

多个主机组里的全部主机：如果想在 一个或多个主机组里的全部主机上创建同一个主机扩展，你可以用 **hostgroup_name** 域。如下将在一系列自 HOSTGROUP1 到 HOSTGROUPN 的主机组里的全部主机上创建同一个主机扩展。如下的主机扩展是同一个(如有相同的联系人组、通知间隔等)。

```
define hostescalation{  
    hostgroup_name          HOSTGROUP1, HOSTGROUP2, ..., HOSTGROUPN  
    other escalation directives  
    ...  
}
```

全部主机：如果你想对你配置文件里的全部主机创建同一个主机扩展，你可以在 **host_name** 域里使用通配符。如下将对你配置文件里的全部主机定义同一个主机扩展。全部的主机扩展是同一个(如有相同的联系人组、通知间隔等)。

```
define hostescalation{
```

```

    host_name                *

    other escalation directives

    ...

}

```

不包含主机：如果在一系列的主机和主机组但不包含某些主机上创建同一个主机扩展，可以在主机或主机组定义前加上 `!` 符号。

```

define hostescalation{

    host_name                HOST1, HOST2, !HOST3, !HOST4, ..., HOSTN

    hostgroup_name

    HOSTGROUP1, HOSTGROUP2, !HOSTGROUP3, !HOSTGROUP4, ..., HOSTGROUPN

    other escalation directives

    ...

}

```

6.2.7. 主机依赖的定义

多个主机：如果想在多个主机上创建同一 [主机依赖](#)，你可以使用 `host_name` 或 `dependent_host_name` 域或同时使用。如下定义将生成六个分离的主机依赖，主机 `HOST3`、`HOST4` 和 `HOST5` 将依赖于 `HOST1` 和 `HOST2`。以上的主机依赖是同一个（如有相同的通知失效处理等）。

```

define hostdependency{

    host_name                HOST1, HOST2

    dependent_host_name      HOST3, HOST4, HOST5

    other dependency directives

    ...

}

```

多个主机组里的全部主机：如果对一个或多个主机组里的全部主机创建同一个主机依赖，你可以用 `hostgroup_name` 或 `dependent_hostgroup_name` 域或两个都用。在如下例中，主机组 `HOSTGROUP3` 和 `HOSTGROUP4` 里的全部主机依赖于主机组 `HOSTGROUP1` 和 `HOSTGROUP2` 的主机。如下的主机依赖同一个只是主机名不同（如有相同的通知失效处理等）。

```

define hostdependency{

    hostgroup_name           HOSTGROUP1, HOSTGROUP2

```

```

dependent_hostgroup_name  HOSTGROUP3, HOSTGROUP4

other dependency directives

...

}

```

6.2.8. 主机组的定义

全部主机：如果你想把你全部的配置文件里的主机都定义在同一个主机组里，你可以在 **members** 域里使用通配符。如下的配置文件里的全部主机都定义到一个叫 **HOSTGROUP1** 主机组。

```

define hostgroup{
    hostgroup_nameHOSTGROUP1
    members          *
    other hostgroup directives
    ...
}

```

6.3. 用户自定义对象变量

6.3.1. 介绍

用户通常想在主机、服务或联系人的对象里加入自己定制的变量，这些变量象 SNMP 共同体名、MAC 地址、AIM 用户名、Skype 帐号和街道名称等等，可能有各种各样的东西无法列完。这样会使 Nagios 不具备通用性并且无法保持一个特定的架构。Nagios 试图更为柔性化，这就意味着需要处理这种情况，例如在 Nagios 的主机对象定义中，“address”是一个 IP 地址也可以是任何东西，只要对使用者而言是个可读可操作的，无论用户怎么设置都行。

但还是有必要在 Nagios 配置文件中提供一种可供管理和保存的处理方法而不是与现有变量域混用的方法。Nagios 试图在对象的定义中引用用户自定义变量来解决这个问题。用户自定义变量的方法可以让用户在主机、服务和联系人对象定义里加入属性，在通知、事件处理和对主机与服务的检测中使用这些变量。

6.3.2. 用户自定义变量的基本规则

使用用户自定义变量需要注意如下几个要点：

- 必须以下划线(_)开头来定义变量名称以防止与标准域名称混淆；
- 自定义变量名是大小写敏感的；
- 自定义变量是可以象一般的变量那样被 [继承](#)传递的；
- 自定义变量名是可以被脚本里引用的，在 [宏和环境变量](#)中有说明。

6.3.3. 例子

这有一个如何在对象中定义不同类型的用户自定义变量的例子：

```
define host{
    host_name      linuxserver
    _mac_address    00:06:5B:A6:AD:AA ; <-- Custom MAC_ADDRESS variable
    _rack_number    R32                ; <-- Custom RACK_NUMBER
variable
    ...
}

define service{
    host_name      linuxserver
    description     Memory Usage
    _SNMP_community public              ; <-- Custom SNMP_COMMUNITY
variable
    _TechContact    Jane Doe           ; <-- Custom TECHCONTACT variable
    ...
}

define contact{
    contact_name     john
    _AIM_username     john16            ; <-- Custom AIM_USERNAME
variable
    _YahooID john32                ; <-- Custom YAHOOID variable
    ...
}
```

6.3.4. 在宏里使用用户自定义变量

在Nagios的检测、通知等的脚本和执行程序里可以引用用户自定义变量，通过使用[宏](#)或是环境变量来实现。

为防止混淆不同对象类型中的用户定制变量，Nagios 在宏和环境变量的名字里，对用户定义的主机、服务或是联系人的变量名之前分别加上了“_HOST”、“_SERVICE”或“_CONTACT”以示区分。下面的表格中给出前面例子中的用户自定义变量在宏和环境变量这中的可引用的命名。

表 6.1.

对象类型	变量名	宏名	环境变量
主机	MAC_ADDRESS	\$_HOSTMAC_ADDRESS\$	NAGIOS__HOSTMAC_ADDRESS
主机	RACK_NUMBER	\$_HOSTRACK_NUMBER\$	NAGIOS__HOSTRACK_NUMBER
服务	SNMP_COMMUNITY	\$_SERVICESNMP_COMMUNITY\$	NAGIOS__SERVICESNMP_COMMUNITY
服务	TECHCONTACT	\$_SERVICETECHCONTACT\$	NAGIOS__SERVICETECHCONTACT
联系人	AIM_USERNAME	\$_CONTACTAIM_USERNAME\$	NAGIOS__CONTACTAIM_USERNAME
联系人	YAHOOID	\$_CONTACTYAHOOID\$	NAGIOS__CONTACTYAHOOID

6.3.5. 用户自定义变量与继承

象标准的主机、服务或联系人对象里的变量一样，用户自定义变量同样可以 [继承](#)。

6.4. 对象继承关系

6.4.1. 介绍

本文件试图解释什么是对象继承和如何在 [对象定义](#) 里使用它。

如果你在前过之后被如何进行递归和继承搞迷糊了，你可以看一下 Nagios 发行包里的简单的对象配置文件。如果还没有帮助，扔个邮件写清楚 [详细情况](#) 描述你的问题到 **nagios-users** 邮件列表。

6.4.2. 基础

对于全部的对象定义说明，有三个变量影响着递归和继承关系，下面用 ^(*) 符号标记说明：

```
define someobjecttype{
    object-specific variables ...

    name                template_name(*)

    use                  name_of_template_to_use(*)

    register [0/1](*)
}
```

第一个变量是 **name**，只是一个可供其他对象定义时提供模板引用名字，以使其他对象可以继承属性和变量。模板名字必须是唯一的且继承者要有相同的类型定义，也就是说，不能给主机对象定义有两个或以上的模板含有同一个主机模板。

第二个变量是 **use**，用来表示对象的属性和变量是继承于哪个指定模板。指定的这个继承来源必须是一个命名过的另一个对象模板(用变量 **name** 确切命名过的)。

第三个变量是 **register**。这个变量用于告知这个对象定义是否需要 Nagios “注册”。默认情况下，对象定义是需要 Nagios 注册。如果你想利用一个对象定义的部分内容作为一个模板，你可以让它不在 Nagios 里注册(后面将提供一个例子)。取值：0 = 不做注册；1 = 注册(默认值)。这个变量是不被继承的；每个对象模板都须明确地将这个 **register** 变量设置为 0。防止 **register** 被设置为 1 的继承后覆盖需要注册的对象定义。

6.4.3. 本地变量和继承变量比较

在理解继承关系时有一个很重要就是本地的对象变量总是优先于模板里的对象变量值，看一下下面的例子中两个主机的定义(没有提供全部的必备变量)：

```
define host{
    host_name          bighost1
    check_command       check-host-alive
    notification_options d,u,r
    max_check_attempts 5
    name               hosttemplate1
}

define host{
    host_name          bighost2
    max_check_attempts 3
    use                hosttemplate1
}
```

你注意到主机 **bighost1** 的定义中引用了模板 **hosttemplate1** 定义，主机 **bighost2** 的定义则使用了主机 **bighost1** 作为模板。一旦由 Nagios 来处理这些数据，那么主机 **bighost2** 相当于是这么定义的：

```
define host{
    host_name          bighost2
    check_command       check-host-alive
    notification_options d,u,r
    max_check_attempts 3
}
```

可以看到 `check_command` 和 `notification_options` 变量从模板 (也就是主机 `bighost1` 的定义) 继承而来, 而 `host_name` 和 `max_check_attempts` 没有从模板对象中继承, 而被限定于本地变量。这应该是一个相当容易理解的概念。

提示



如果你想让本地串变量继承来自于对象模板的定义, 其实你可以这么干, 看一下 [下面](#) 的内容讲解。

6.4.4. 继承关系链

对象可以从多层次地使用模板对象的属性和变量 (儿子可以引用老爸的老爸的东西, 但更象老爸), 如下例:

```
define host{
    host_name          bighost1
    check_command       check-host-alive
    notification_options d,u,r
    max_check_attempts 5
    name               hosttemplate1
}

define host{
    host_name          bighost2
    max_check_attempts 3
    use                hosttemplate1
    name               hosttemplate2
}

define host{
    host_name          bighost3
    use                hosttemplate2
}
```

注意主机 `bighost3` 变量来自主机 `bighost2` 中定义, 而其后是继承主机 `bighost1` 的内容。采用如此方式来处理配置数据, 其结果就象下面的主机定义一样:

```
define host{
    host_name          bighost1
```

```

        check_command      check-host-alive
        notification_options d,u,r
        max_check_attempts 5
    }

define host{
    host_name      bighost2
    check_command  check-host-alive
    notification_options d,u,r
    max_check_attempts 3
}

define host{
    host_name      bighost3
    check_command  check-host-alive
    notification_options d,u,r
    max_check_attempts 3
}

```

对于对象继承层次的深度没有限度（老爸的老爸的老爸的...没有尽头的），但你为了保持清楚的定义以便于维护的话可能需要减少继承的层次（别把老祖宗也抬出来，家谱没办法画啦！:-D）。

6.4.5. 用不完整的对象定义做模板

用定义不完整的对象定义来做对象模板给其他对象做继承源是可以的，“不完整”的对象意思是定义了对象不含全部内容的对象。使用不完整的对象来做模板这可能看起来很奇怪，但却推荐你这么做，为什么呢？因为它可以定义一堆默认的对象属性给其他的对象用于继承（这就象介绍父子俩：老爸长得的五官很端正...，儿子象他爸）。看下面的例子：

```

define host{
    check_command      check-host-alive
    notification_options d,u,r
    max_check_attempts 5
    name               generichosttemplate
    register            0
}

```

```

define host{
    host_name          bighost1
    address             192.168.1.3
    use                 generichosttemplate
}

define host{
    host_name          bighost2
    address             192.168.1.4
    use                 generichosttemplate
}

```

注意到第一个主机对象的定义是不完整的，因为它缺少了必须的 **host_name** 变量。我们不想定义这个 **host_name**，因为它是一个通用的对象模板。为了防止它被 Nagios 理解为一个一般的主机，我们把 **register** 变量设置为 0。

主机 **bighost1** 和 **bighost2** 的定义来自于通用对象模板的继承。我们只是选择性地覆盖了 **address** 变量定义。也就是说，这两个主机将有相同的属性，除了 **host_name** 和 **address** 变量不一样。在 Nagios 处理这个样例中的配置数据时将等同于做如下对象的定义：

```

define host{
    host_name          bighost1
    address             192.168.1.3
    check_command       check-host-alive
    notification_options d,u,r
    max_check_attempts 5
}

define host{
    host_name          bighost2
    address             192.168.1.4
    check_command       check-host-alive
    notification_options d,u,r
    max_check_attempts 5
}

```

不完整的对象定义的优势最少最少的一点就是你可以在对象定义的时候少打很多字母，同样，它也可以在你改变大量的主机的变量定义时减少你的痛苦。（——原作者无非是想让用户尽量在对象定义的时候用这种理性的表达方式，而不是一团数据的粘贴来做）

6.4.6. 用户定义变量

任何你想在主机、服务或联系人等的带有 [用户定制变量](#) 的模板定义将象标准的对象变量一样做对象继承的传递（介绍一对特殊的父子：老爸长得高过姚明，儿子也很高），象下面的例子：

```
define host{
    _customvar1          somevalue          ; <-- Custom host variable
    _snmp_community      public              ; <-- Custom host variable
    name                  generichosttemplate
    register              0
}

define host{
    host_name             bighost1
    address                192.168.1.3
    use                    generichosthosttemplate
}
```

主机 **bighost1** 将会继承来自于模板 **generichosttemplate** 的用户定义变量 **_customvar1** 和 **_snmp_community** 和各自的值。其结果是主机 **bighost1** 的定义就象这样：

```
define host{
    host_name             bighost1
    address                192.168.1.3
    _customvar1           somevalue
    _snmp_community       public
}
```

6.4.7. 取消继承的字符串值

有些情况下，你并不想让你的主机、服务或联系人对象定义继承从模板里定义的值，在是这种情况下，你可以指定为“*null*”（是不带双引号的）做为变量的值以防止继承模板的值（介绍父子俩：老爸个子高过姚明，但儿子很普通，儿子多高还是不知道吧？！），如下面的例子：

```
define host{
```

```

        event_handler      my-event-handler-command
        name                generichosttemplate
        register           0
    }

define host{

        host_name          bighost1
        address            192.168.1.3
        event_handler      null
        use                generichosthosttemplate
    }

```

在上例中，主机 **bighost1** 的对象定义将不再继承 **event_handler** 变量，而这个变量是定义在模板 **generichosttemplate** 之中。其结果就是主机 **bighost1** 的定义是下面这样子：

```

define host{

        host_name          bighost1
        address            192.168.1.3
    }

```

6.4.8. 继承时附加字符串值

Nagios 在处理时总是让本地变量高于从模板继承，但有些时候想让本地变量与继承模板的对象同时起效。

这种“*附加继承*”式的继承可以是在本地变量中用一个附加(也就一个“+”号)式来表示它。但这种特性只支持标准（非用户定制）变量中包含这种串定义(介绍父子俩：老爸个子是二米一，儿子个子比老爸高出两公分)。如下面的例子：

```

define host{

        hostgroups          all-servers
        name                generichosttemplate
        register           0
    }

define host{

        host_name          linuxserver1
        hostgroups          +linux-servers,web-servers
    }

```



```

use
    generichosttemplate
}

```

在上面例子中，主机 `linuxserver1` 的本地变量 `hostgroups` 将会附加在由模板 `generichosttemplate` 的变量之上，其主机 `linuxserver1` 的结果就是：

```

define host{
    host_name          linuxserver1
    hostgroups         all-servers,linux-servers,web-servers
}

```

6.4.9. 隐含继承

通常情况下，你必须清晰地指定哪些对象的变量是从模板继承的，有很少的情况并不遵守这个规则，也就是当 Nagios 认为你想利用其中的一个值而不是从相关对象引用时是这样的。例如，如果你不指明晰地指定有些服务的变量将是主机与服务的结合中获得。

下表列举了这些情况。当你没有特别清晰地指定对象变量值并且没有可从模板继承的值的时候，下面列出的情况就会从相关对象里面引用从而实现隐含继承。

表 6.2.

Object Type	Object Variable	Implied Source
服务	<code>contact_groups</code>	绑定的主机对象中的 <code>contact_groups</code> 域
	<code>notification_interval</code>	绑定的主机对象中的 <code>notification_interval</code> 域
	<code>notification_period</code>	绑定的主机对象中的 <code>notification_period</code> 域
主机扩展	<code>contact_groups</code>	绑定的主机对象中的 <code>contact_groups</code> 域
	<code>notification_interval</code>	绑定的主机对象中的 <code>notification_interval</code> 域
	<code>escalation_period</code>	绑定的主机对象中的 <code>notification_period</code> 域
服务扩展	<code>contact_groups</code>	绑定的服务对象中的 <code>contact_groups</code> 域
	<code>notification_interval</code>	绑定的服务对象中的 <code>notification_interval</code> 域
	<code>escalation_period</code>	绑定的服务对象中的 <code>notification_period</code> 域

6.4.10. 在对象扩展里的隐含与附加继承

服务扩展与服务扩展的对象定义可以将隐含继承和附加继承结合起来使用。如果对象扩展里不继承其他扩展对象模板中 `contact_groups` 或是 `contacts` 域的值,而且它 `contact_groups` 或 `contacts` 域里以 (+) 号开头,那么,主机或服务定义里的 `contact_groups` 或 `contacts` 域将使用附加继承逻辑的规则来处理。

搞迷糊了吧? 这有个例子:

```
define host{
    name                linux-server
    contact_groups      linux-admins
    ...
}

define hostescalation{
    host_name           linux-server
    contact_groups      +management
    ...
}
```

上面的例子相当于这样:

```
define hostescalation{
    host_name           linux-server
    contact_groups      linux-admins,management
    ...
}
```

(——如果你觉得这是个怪里怪气的规则,还是老实地写明白的好)

6.4.11. 多重继承

迄今为止,所有的例子都是从单一的源上来做对象定义时继承对象的变量或域值。你可以在一个复杂的配置里使用多个源来完成对象的变量或域值的定义。象下面的例子:

```
# Generic host template

define host{
    name                generic-host
    active_checks_enabled 1
    check_interval      10
```

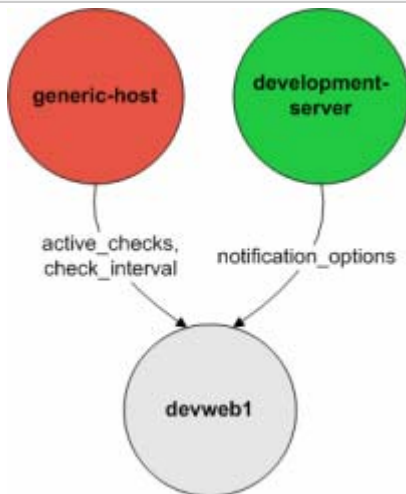
```

...
    register                0
}

# Development web server template
define host{
    name                    development-server
    check_interval          15
    notification_options    d,u,r
    ...
    register                0
}

# Development web server
define host{
    use                    generic-host, development-server
    host_name              devweb1
    ...
}

```



上例中，主机 **devweb1** 是从两个源模板 **generic-host** 和 **development-server** 中继承变量和域。注意到 **check_interval** 域在两个源里都有定义。由于 **generic-host** 是第一个被主机 **devweb1** 的 **use** 域里说明的模板，那么它的 **check_interval** 域值将传给主机 **devweb1**。那么这种继承规则下，主机 **devweb1** 将象如下的定义：

```

# Development web server

```

```

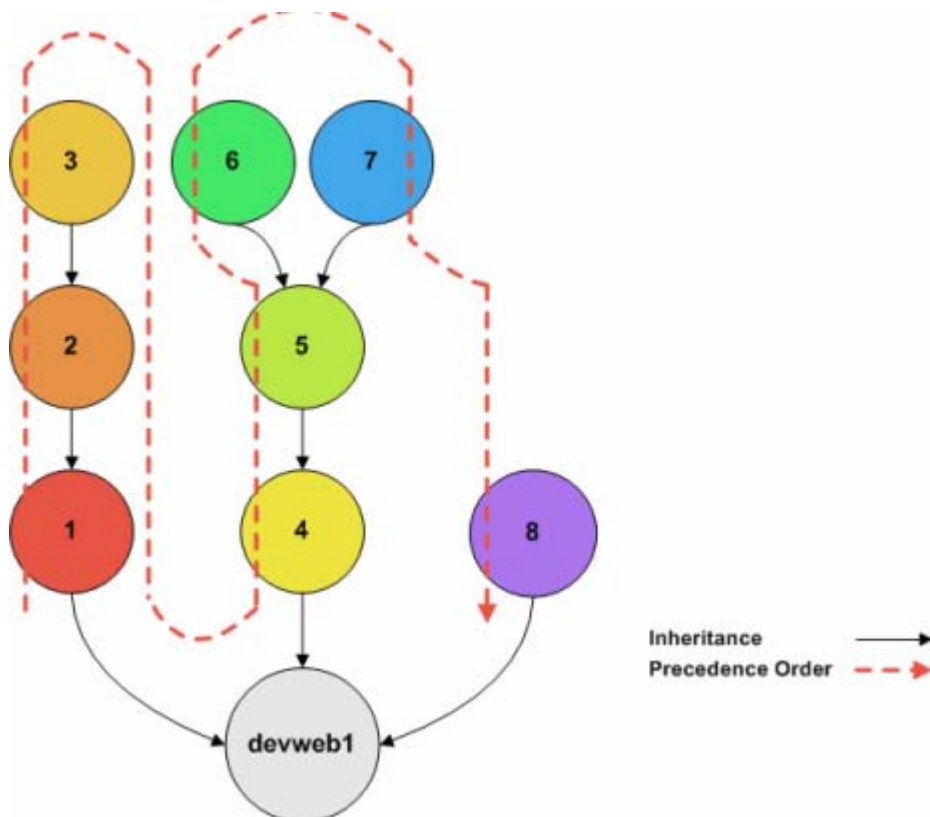
define host{
    host_name          devweb1
    active_checks_enabled 1
    check_interval      10
    notification_options d,u,r
    ...
}

```

6.4.12. 在多重继承中指定优先级

当你使用多个源做继承时，告诉 Nagios 如何处理那些变量是很重要的事。一般是 Nagios 将会使用 **use** 域中指定的第一个对象模板（就是第一个源）。既然是可以从多个源里来继承变量或域值（——尤其是每个源都是多层次继承下来的时候），有必要清晰地处理这些变量和域的优先级别。

考虑如下的涉及到三个对象模板的主机定义：



```

# Development web server
define host{
    use          1, 4, 8
    host_name    devweb1
    ...
}

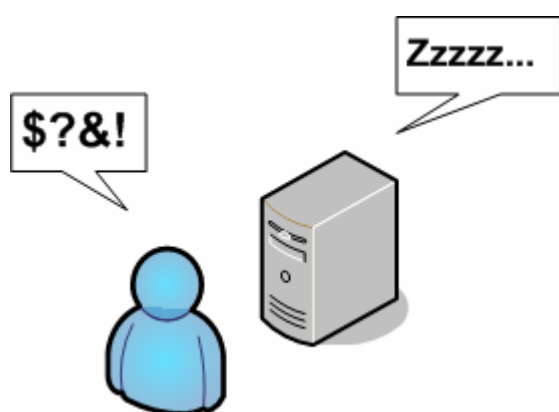
```

}

如果从一个或多个模板中要继承而涉及到多个对象的引用时，优先级的处理方式是以右侧优先（——就是 use 中指明的第一个直接对象源 1、直接源的父对象节点 2、对象节点 2 的父对象 3、第二个直接对象源 4、源 4 的父节点 5... 依次类推，注意看图中的示意）。测试、检验和排错将有助于你更准确地理解象这种复杂的继承关系。（——老婆，跟我一块儿出来看上帝...）

6.5. 计划停机时间

6.5.1. 介绍



Nagios 里可以给所监控主机与服务指定一个计划的停机时间。这在得知所监控的服务或主机要在某个时间内要停机以升级等时候非常有用。

6.5.2. 计划停机时间

可以用 [扩展信息CGI模块](#) 来对某主机或服务指定计划停机时间(可以在查看主机或服务信息时来做)。点击一下“给此主机/服务设置计划停机时间”的链接来开始编制一个计划停机时间。

一旦给主机与服务编制了一个计划停机时间，Nagios 将会给主机与服务加入一条注释以说明在这个期间该主机与服务是处于计划停机时间内。当计划停机时间过去了，Nagios 将自动地删除那条添加的注释。很棒吧？

6.5.3. 固定的与可变的停机时间

当通过 Web 来编制一个主机与服务的计划停机时间时，Nagios 会询问停机时间是固定式还是可变式，这里来解释一下“固定式”与“可变式”有何不同：

“固定式”停机时间启动和停止在你所编制计划所设定的时间内开始与结束，这当然很简单啦...

“可变式”停机时间可以用在当知道主机与服务要停机 X 分钟(或 X 小时)但是并不知道什么时候开始停机时，当使用可变式停机时间，Nagios 将在某个时间开始执行停机，到你指定的时间间隔达到后结束停机。它假定了主机与服务使用一个可变的停机时间段来做停机时的操作，而这个停机时间段开始于主机进入宕机(或不可达)状态或是服务处于非正常状态时，结束时间是经过了你指定的时间间隔之后的那个时间点，

即便是在此之前主机与服务已经恢复也是认为是它还处于停机时间内。对于这样的情况你将很需要这种停机时间定义，你需要做一个故障修复，但需要重新启动机器才能让它真正启效。很聪明，不是吗？

6.5.4. 触发停机时间

当编制主机与服务的停机时间时需要给出可对它“触发”的停机时间。什么是触发停机时间？有触发的停机时间开始于编制时所指定的停机时间开始的时刻，这对于很多个主机与服务的停机时间开始于编制好的某个停机时间条目时是非常有用的。比如，当编制一个主机的停机时间(因需要做维护而做停机)时，需要在网络拓扑中针对这个主机的全部子节点主机定制触发停机时间。

6.5.5. 计划停机时间对通知产生什么影响？

当主机与服务处于停机时间内时，Nagios 将不会送出针对这个主机与服务的一般意义的通知。但是，会送出一条停机时间开始“DOWNTIMESTART”的通知，这将给主机与服务的管理者一个提示，在此之后将不会收到主机与服务故障时的告警通知直到停机时间结束。

当主机与服务的停机时间结束时，Nagios 将再次可以送出针对这个主机与服务的一般意义的通知，也会送出一条停机时间结束“DOWNTIMEEND”的通知，这将给主机与服务的管理者提醒，在此之后会再次收到各种该有的通知了。

如果预置的停机时间被提前取消(在期满之前)，会送出一条停机时间取消“DOWNTIMECANCELLED”的通知给相关的管理员。

6.5.6. 计划停机时间的重叠

这就好象是“天啊，它又没动静了。”的并发症，你知道我在说什么。你编制了一个服务停机时间来做“例行”的硬件升级，只是在此之后才意识到操作系统的驱动不支持它！硬盘 RAID 搞掉了或是驱动映像失败或是原始盘已经彻底完蛋了。象这样的故事会发生在任何一个你认为只是“例行”的停机时间里，而且相似的故事会一幕一幕地重演着。

看下面这个场景：你是个做网管的倒霉蛋，而且

- 你给主机 A 定制了停机时间是每周一晚上 19:30-21:30；
- 通常大约是在周一晚上 19:45 时会开始硬件升级；
- 在一个很不幸日子里，你在浪费了一个半小时来处置 SCSI 和驱动不兼容之后，机器终于开启了；
- 在到了晚上 21:15 时，你才发现一个分区无法挂接或是在盘上怎么也找不到它；
- 知道要搞很长时间了，你不得不返回重编制对主机 A 编制一个额外停机时间，从周一晚上 21:20 到周二凌晨 1:30；

如果你给主机与服务编制了重叠的计划停机时间(在上例中，有 19:40 到 21:30 和 21:20 到 1:30 两个停机时间)时，Nagios 将会等待，直至最后一个编制的停机时间结束时才会送出相关的通知。在上例中，直到周二早晨的 1:30 之前的这段时间里，主机 A 的各种通知一直会被压制着。

6.6. 时间周期

或许是...“正当其时?”

6.6.1. 介绍

[时间周期](#)对象定义可用于控制何时各种不同的监控与报警的逻辑可以执行或操作。例如可以限定：

- 何时可以执行对主机与服务的计划任务检测；
- 何时可以送出通知；
- 何时应用通知扩展；
- 何时依赖关系是正确的；

6.6.2. 时间周期中的优先权

[时间周期](#)的对象定义中有多个不同类型的域，包括周计划、月计划、日历型日期。不同类型的域有不同的优先级别而且会覆盖同一个时间周期定义里的其他域值。不同类型的域的优先级从高到低依次如下（——后面是译者加的例子）：

- 日历型日期(2008-01-01)——指定奥运会开幕的那天是(2008-08-08)
- 指定月份的日期(January 1st)——国庆是每年的十月一日——(October 1st)
- 一般月份里的日期(Day 15)——每个月的 5 号发工资啊(Day 5)
- 指定月份里的星期几的次数(2nd Tuesday in December)——父亲节是每年六月的第三个星期天(3th Sunday in June)
- 指定星期几的次数(3rd Monday)——每隔四周的周六都要执班(4rd Saturday)
- 一般的周计划(Tuesday)——每周六和周日都可以休息(Saturday Sunday)

不同的时间周期域的样例可以查阅 [这篇](#) 文档。

6.6.3. 时间周期在主机与服务检测时是如何起作用的？

主机与服务定义里的可选域 **check_period** 可用于控制限定特定的时间周期，它可以用于控制何时进行规格化的计划任务，何时做自主检测等。

如果没有在 **check_period** 域来指定一个时间周期，Nagios 将在任何需要的时候执行计划性的自主检测，实际上相当于设置一个 24x7 的时间周期。

Specifying a timeperiod in the 在 **check_period** 域里指定一个时间周期可以限定 Nagios 执行规格化计划检测的时间，主机与服务自主检测的时间。当 Nagios 尝试去对主机或服务进行一个规格化计划表检测时，它将确保下次检测是在指定的合法时间段内进行。如果不是，Nagios 将调整下次检测时间以使下次检测处于指定的时间周期所限定的合法时间内，这意味着主机或服务的检测可能在下个小时、下一天或下一周等等的时间里不会检测直至到时间。

注意



按需检测和强制检测将不受 `check_period` 域所指定的时间周期的限制，这个时间周期只是对规格化计划执行的自主检测做限制。

强烈建议你对全部的主机与服务使用 24x7 这个时间周期，除非你有一个明确的理由可以不这样做。如果没有用 24x7，可能在你指定时间周期的非合法时间里(无监控的黑色时间段)将会有些麻烦：

- 主机与服务状态将不再改变；
- 联系人将几乎不会收到主机与服务重置报警；
- 如果主机与服务从故障中恢复，所属的联系人将不会立即收到恢复的通知。

6.6.4. 时间周期在联系人通知时是如何起作用的？

通过使用主机与服务对象定义里的 `notification_period` 域可以指定一个特定的时间周期，它可以限定 Nagios 主机与服务在认定故障或故障恢复时送出通知。当主机的通知将要被送出时，Nagios 将会确保当前时刻处于 `notification_period` 指定的时间周期里是合法的时间。如果是合法时间，Nagios 将尝试对每一个联系人送出故障与恢复的通知。

也可以用多种时间周期来控制通知通向不同的联系人。指定 [联系人对象定义](#) 里的 `service_notification_period` 和 `host_notification_period` 域，可以对每个联系人指定一个“按应需求”的时间周期。每个联系人将只是在指定的时间周期里才会收到主机与服务的通知。

如何创建一个“按应需求”循环的例子可以查阅 [这篇](#) 文档。

6.6.5. 时间周期在通知扩展里是如何起作用的？

使用服务与主机的 [通知扩展对象](#) 定义里的可选项 `escalation_period` 域可以指定一个特定时间周期，它将限定在哪个时间内是扩展项是合法的且可用的。如果没有使用在扩展对象里的 `escalation_period` 域，那么扩展对象将认定所有时间都是合法时间。如果使用了 `escalation_period` 域来指定时间周期，Nagios 将只是在指定时间周期所限定的合法时间内使用扩展对象。

6.6.6. 时间周期在依赖关系里是如何起作用的？

通过使用主机与服务的 [依赖](#) 关系对象里的可选项 `dependency_period` 域来指定一个时间周期，它可以限定依赖关系对象在哪个时间段内是合法的且可以使用。如果没有在依赖关系对象里使用 `dependency_period` 域，依赖关系对象在任意时间里都是合法可用的。如果在对象依赖关系里的 `dependency_period` 域指定了时间周期，Nagios 将只是在指定时间周期所限定合法时间内使用该依赖对象。

6.7. 通知

6.7.1. 介绍

我收到很多关于通知如何运作更精确的问题。此处将尝试解读何时和如何将主机与服务通知送出以及谁会接收这些通知。

通知扩展的解释在 [这篇](#) 文档。

6.7.2. 何时会做通知?

送出通知的判定是由主机与服务的检测逻辑来完成的。主机与服务的通知发生于如下情形:

- 当一个硬态状态变化时; 更多有关状态类型与硬态变化的内容请查阅 [这篇](#) 文档。
- 当主机或服务仍旧处于一个硬态的非正常状态而且最后一次通知送出的时间超过了主机与服务对象定义里的<notification_interval>域所指定的时间时。

6.7.3. 谁会收到通知?

每个主机与服务对象定义里都有<contact_groups>域来指定接收此主机与服务通知内容的联系人组。联系人组可以包括一个或几个相互独立的联系人。

当 Nagios 送出主机与服务的通知, 将会通知每个联系人组里的联系人成员, 联系人组是由对象定义里的<contactgroups>域来设定。Nagios 实现了联系人可以属于多个联系人组, 所以会在做通知之前将联系人组里重复出现的联系人去掉保证每个联系人收到有且只有一次通知。

6.7.4. 送出通知时必须要通过什么样的过滤器?

因为并非每一个接收送出通知的联系人都需要收到通知所以需要过滤器来处理它。通知送出前有好几个经过的过滤器, 正因如此, 指定有联系人就可能收不到信息因为过滤器可能把它要收到的信息组过滤掉了。下面稍详细点地介绍一下通知在送出前要通过的过滤器...

6.7.4.1. 程序层面的过滤器

首先必须通过的过滤器是在程序里面内嵌是否发送通知的过滤器。它由主配置程序里的 [enable_notifications](#) 变量值初始化, 但可在运行时通过Web接口改变它。如果通知在程序层面里是不使能的, 那么在这期间里, 不会送出任何主机与服务的通知。如果使能了它, 仍旧有其他的过滤器要通过...

6.7.4.2. 主机与服务过滤器

主机与服务通知要通过的第一个过滤器是检查主机与服务是否处于 [计划停机时间定义](#) 的时间段内。如果在停机时间段内, 联系人不会收到通知。如果不是在停机时间段内, 通知会通过这个过滤器而到下一个过滤。额外的提醒是, 如果是在主机的停机时间段内, 给主机上的服务通知将会被压制。

要通过的第二个过滤器是在检查主机与服务是否处于 [抖动](#) (如果你使能了感知抖动检测项的话)。如果服务或主机当前处于抖动, 联系人不会收到通知, 其他情况下, 这个过滤会通过进入到下个过滤器。

要通过的第三个过滤器是给主机的与服务的通知选项。每个服务对象定义含有一个选项过滤以决定是否在报警、紧急和恢复等状态时送出通知。相似的, 主机对象定义里含有选项以决定是否在宕机、不可达和恢复等状态时送出通知。如果主机与服务的通知没有通过这些过滤选项, 那么联系人不会收到通知, 如

果通过了，则会进入下一个过滤... 注意，主机与服务的恢复通知仅仅是当诱发它的原始故障通知也送出时才会送出，这样就不会收到一条不知道原因的故障恢复通知的。

要通过的第四个过滤器是给时间周期的检查。每个主机与服务对象定义里都有一个 `<notification_period>` 通知时间周期选项来指定何时送出通知是合法的时间。如果送出通知的时间没有落在指定的时间周期所划定的范围内的话，没有人会收到通知。如果时间是处于指定的时间周期之内的话，该过滤会通过，则会进入下一个过滤... 注意：如果时间周期的过滤器没有通过的，Nagios 将会重新编制该主机与服务（如果它处于非正常状态的话）的通知送出时间，使送出时间处于合法的时间周期规定。这将有助于保证联系人在下一个时间周期到来时尽可能早地收到故障通知。

最后一个主机与服务的过滤器是由两个要素条件控制：(1) 针对该主机与服务的已经送出的最后一条通知所发出的时间；(2) 主机与服务在最后一通知发出后仍旧处于相同的非正常状态所处的时间长度。如果遇到这两个限定条件，Nagios 将会用最后一次通知送出时间到当前时间的时段来比对主机与服务对象定义里的 `<notification_interval>` 通知间隔域，看看是否到达或超出。如果还没有到通知间隔所设置的时段，不会送出通知给任何人。如果这个时段已经超出了间隔设置而且第二个条件不成立的话（就是说因为状态不一样而送出通知），通知就会被送出！是否真正地送出通知，还必须要通过每个联系人的过滤器控制...

6.7.4.3. 联系人过滤器

在这个点上，通知过程已经通过了程序过滤和全部的主机与服务对象里所设置的过滤，开始通知 [每一个它该通知到的联系人](#)。这是否就意味着要每个联系人都会收到通知呢？并不是这样！每个联系人都有各自的联系人过滤器，通知要经过这些过滤后才能收到通知。注意：联系人过滤器指定给每一个联系人但不会影响到其他联系人是否收到通知。

第一个联系人过滤器是联系人对象定义里的有关主机的或服务的过滤通知选项。每个联系人可以指定出对于服务，是否要收到告警状态、紧急状态和恢复状态的通知，同样地，也可以指定针对主机是否要收到主机宕机、变为不可达或是恢复的通知。如果这些在联系人里的主机和服务的过滤没有通过的话就不会收到通知，如果设置了要送出通知，那么会进入下一个过滤器... 注意：只是那些针对于主机与服务的原始故障而产生的通知才会送出，不会有人收到一个没有故障原因通知却有状态恢复的通知...

最后一个过滤是联系人里的时间周期设置的检查。每个联系人对象定义里的 `<notification_period>` 通知接收时间周期域指定了联系人可以接收通知的时间周期。如果通知的时间没有落入指定的时间周期的时段内，联系人不会收到通知。如果在合法的时段区间里，联系人会收到通知！

（译者注：数一数，一共有七个过滤器！第 1 个是总阀门，第 2 到第 5 个是针对服务与主机状态的，后面 2 个是针对每个联系人的，很复杂，但是提供了很大的控制度）

6.7.5. 通知的方式

对于故障与恢复的通知方式，Nagios提供了多种供选择：BP机、蜂窝电话、电子邮件、即时信息、警报声音、电击(这是个什么东西?)等等。如何送出通知将依赖于你的 [对象定义文件](#)里的 [通知命令](#)。

注意



如果你是按照 [快速安装指南](#)来安装的Nagios的话，它将配置成用EMail送出通知。你可以在这个配置文件里找到并查看对应EMail送出通知的命令 `/usr/local/nagios/etc/objects/commands.cfg`。

特定的通知方式(象 BP 机等)并没有直接融合在 Nagios 代码中因为这没有必要。Nagios 的核心设计思想并不是把 Nagios 搞成一个集成完整统一的一个应用程序(all-in-one)。如果这种服务嵌入到 Nagios 的核心之中将会使得用户很难加入自己的检测方法，而且修改检测等等也不方便。通知的处理也是如此。有成百上千种方式来实现检测与通知，因而为何要舍近求远呢？最好的方式是提供一个外部调用的入口(如一个执行脚本或一个成熟的消息系统)来做这种杂事。有一些消息处理包或是蜂窝电话挂件的资源可以处理通知，在下面一节里给出了列表。

6.7.6. 通知类型的宏

当编写通知命令时，需要理解是什么通知类型产生的。那个 [\\$NOTIFICATIONTYPE\\$](#)宏将用一个字符串来指出是哪个类型。下表列出这个宏可能的值以及相关的描述信息：

表 6.3. 通知类型的宏

值	描述
PROBLEM	服务与主机刚刚(或是仍旧)处于故障状态。如果收到服务通知，可能服务是处于告警、未知或是紧急状态之中，如果收到是主机通知，主机可能是处于宕机或不可达状态之中
RECOVERY	服务与主机已经恢复。如果是一个服务通知，说明服务刚回到正常状态，如果是主机通知，说明主机刚刚回到运行状态
ACKNOWLEDGEMENT	这是一个主机与服务故障的确认通知。由联系人给特定的主机与服务通过 Web 来初始化一个确认通知
FLAPPINGSTART	主机与服务刚开始处于 抖动
FLAPPINGSTOP	主机与服务刚结束 抖动
FLAPPINGDISABLED	主机与服务刚因为检测抖动被关闭而停止 抖动 ...

值	描述
DOWNTIMESTART	主机与服务刚进入到一个 计划停机时间周期 ，在此后通知会被抑制
DOWNTIMESTOP	主机与服务刚结束了 计划停机时间 。有关故障的通知将恢复
DOWNTIMECANCELLED	给主机与服务所指定的 计划停机时间 刚刚取消。有关故障的通知将恢复

6.7.7. 有用的资源

在 Nagios 中可以配置多种送出通知的方式。这取决于你所想用的方式方法。一旦安装好必须的支持软件并在配置文件里给定了通知命令就可以运用它们了。可行的方式这里只给出几种：

- 电子邮件 (Email)
- BP 机 (Pager)
- 蜂窝电话短信息 (CellPhone SMS)
- Windows 弹出消息 (WinPopup message)
- 各种即时信息 (Yahoo, ICQ, or MSN instant message)
- 声音警报 (Audio alerts)
- 等等...

所有这些全是基于你用通知命令格式来编写了一个命令行。

如果想找一个替代电子邮件送出通知的方法，如用 BP 机或蜂窝电话，查看一下如下软件包。这些可以与 Nagios 结合当故障产生时用一个 Modem 送出通知，这在 EMail 无法送出通知时起作用 (注意，电子邮件在网络出现故障时可能不会送出电子邮件) 我没有真正测试过这些包，但其他人报告说是可以用的...

- [Gnokii](#) 一个手机短信的软件包 (SMS software for contacting Nokia phones via GSM network)
- [QuickPage](#) 数字 BP 机的软件 (alphanumeric pager software)
- [Sendpage](#) BP 机软件 (paging software)
- [SMS Client](#) 给 BP 机或手机发短信的命令行工具 (command line utility for sending messages to pagers and mobile phones)

如果想试验非传统的通知方式，比如说想费时费力地使用声音警报，在你的监控主机上使用合成声音来演绎出你的故障通知，可以迁出 [Festival](#) 项目，如果想用一个独立的声音报警盒子，可以迁出 [Network Audio System \(NAS\)](#) 和 [rplay](#) 项目。

6.8. 事件处理

6.8.1. 介绍



事件处理是一些可选的系统命令(脚本或执行程序)，一旦主机与服务状态发生变化时就会运行它们。

一个明显的例子是使用事件处理来在任何人收到通知之前由 Nagios 来做一些前期故障修复。如下的情况也可能会用到：

- 重新启动一个失效的服务；
- 往协助处置系统里敲入一个故障票；
- 把事件信息记录到数据库中；
- 循环操作主机电源*
- 等等

*循环操作主机电源是个故障处理经验，它是个不容易实现的自动化脚本。在用自动化脚本实现之前要考虑到它的后果。：-)

6.8.2. 何时执行事件处理？

事件处理将会执行，当一个主机或服务处于如下情况时：

- 处于一个软态故障状态时
- 初始进入一个硬态故障时
- 从软态或硬态的故障状态中初始恢复时

状态类型的软态与硬态在 [这篇](#)文档中有详细说明。

6.8.3. 事件处理类型

有几种不同的事件处理类型可以用于主机与服务状态变换的事件处理中：

- 全局主机事件处理
- 全局服务事件处理
- 特定主机事件处理
- 特定服务事件处理

全局主机和服务事件处理将于**每一个**主机和服务状态变更发生时候运行，且稍稍早于特定主机与服务的事件处理。可以用主配置文件里的 [global_host_event_handler](#)和 [global_service_event_handler](#)域来设置全局的主机与服务事件处理命令。

不同的主机与服务可以有各自不同事件处理来处置状态变化，是用 [主机](#)和 [服务](#)对象定义里的 `event_handler`域来指定事件处理命令。这些设置的特定主机与服务的事件处理命令将会在全局主机与服务事件处理运行之后运行。

6.8.4. 使能事件处理

事件处理在程序层面上可通过主配置文件里的 [enable_event_handlers](#)来控制打开或关闭。

特定主机的和服务的事件处理可用 [主机](#)和 [服务](#)对象里的`event_handler_enabled`域来开关。如果全局的 [enable_event_handlers](#)域是关闭的，那么特定主机的和服务的事件处理也不会运行。

6.8.5. 事件处理的执行次序

正如前面所说明的那样，全局的主机与服务的事件会早于主机的和服务的特定的事件处理命令执行。

对于硬态故障和恢复状态的事件处理命令是在通知送出后立即执行。

6.8.6. 编写事件处理命令

事件处理命令可以是SHELL或是Perl程序，同样可以是任意类型语言编写的在命令行下可执行的程序。至少脚本要处理在参数行里处理如下 [宏](#)：

对服务的：[\\$SERVICESTATE\\$](#)、[\\$SERVICESTATETYPE\\$](#)和 [\\$SERVICEATTEMPT\\$](#)；对主机的：[\\$HOSTSTATE\\$](#)、[\\$HOSTSTATETYPE\\$](#)和 [\\$HOSTATTEMPT\\$](#)。

脚本须检测这些作为命令参数传入的值并采取任何必要动作来处理这些值。最好的理解事件处理如何工作的途径是看例子，幸运的是 [下面](#)就提供个例子。

提示



额外的事件处理脚本的例子可以在Nagios发行包的`contrib/eventhandlers/`子目录里找到。有些脚本示范了运用 [外部命令](#)来实现一个 [冗余式](#)和 [分布式](#)监控环境。

6.8.7. 事件处理命令的权限

事件处理命令通常是与运行于本机上的 Nagios 程序的权限是相同的。这可能会有问题，如果你想写成一个用于系统服务重启的命令，它需要有 root 权限以执行一系列命令与任务。

较理想的是让事件处理拥有它将要执行的系统命令所需权限相同的权限。你或许尝试用 [sudo](#)命令来实现它。

6.8.8. 服务事件处理的例子

下面例子给出了监控本机上的 HTTP 服务且在 HTTP 服务对象里指定了 **restart-httpd** 来做为事件处理命令。同样地，假定已经设置了服务对象的 **max_check_attempts** 值为 4 或是大于 4 的值(服务将检测 4 次之后才认定它真的出问题)。该样例服务对象的定义片段象下面这样子：

```
define service{
    host_name                somehost
    service_description      HTTP
    max_check_attempts        4
    event_handler             restart-httpd
    ...
}
```

一旦对服务对象定义了事件处理，必须要保证命令可执行。一个 **restart-httpd** 命令的样例见下。注意在命令行里给命令脚本传递了几个宏——这个很重要！

```
define command{
    command_name      restart-httpd
    command_line      /usr/local/nagios/libexec/eventhandlers/restart-httpd
    $SERVICESTATE$ $SERVICESTATETYPE$ $SERVICEATTEMPT$
}
```

现在，写一个实现的事件处理脚本(它是 **/usr/local/nagios/libexec/eventhandlers/restart-httpd** 脚本文件的内容)。

```
#!/bin/sh

#
# Event handler script for restarting the web server on the local machine
#
# Note: This script will only restart the web server if the service is
#       retried 3 times (in a "soft" state) or if the web service somehow
#       manages to fall into a "hard" error state.
#
#
# What state is the HTTP service in?
```

```

case "$1" in
OK)

    # The service just came back up, so don't do anything...

    ;;

WARNING)

    # We don't really care about warning states, since the service is probably still
running...

    ;;

UNKNOWN)

    # We don't know what might be causing an unknown error, so don't do anything...

    ;;

CRITICAL)

    # Aha! The HTTP service appears to have a problem - perhaps we should restart the
server...

    # Is this a "soft" or a "hard" state?
    case "$2" in

        # We're in a "soft" state, meaning that Nagios is in the middle of retrying the
        # check before it turns into a "hard" state and contacts get notified...

        SOFT)

            # What check attempt are we on? We don't want to restart the web server on
the first

            # check, because it may just be a fluke!
            case "$3" in

                # Wait until the check has been tried 3 times before restarting the web server.
                # If the check fails on the 4th time (after we restart the web server), the
state

                # type will turn to "hard" and contacts will be notified of the problem.

```



```

        # Hopefully this will restart the web server successfully, so the 4th check
will
        # result in a "soft" recovery.  If that happens no one gets notified because
we
        # fixed the problem!
    3)
        echo -n "Restarting HTTP service (3rd soft critical state)..."
        # Call the init script to restart the HTTPD server
        /etc/rc.d/init.d/httpd restart
        ;;
    esac
;;

# The HTTP service somehow managed to turn into a hard error without getting fixed.
# It should have been restarted by the code above, but for some reason it didn't.
# Let's give it one last try, shall we?
# Note: Contacts have already been notified of a problem with the service at this
# point (unless you disabled notifications for this service)
HARD)
    echo -n "Restarting HTTP service..."
    # Call the init script to restart the HTTPD server
    /etc/rc.d/init.d/httpd restart
    ;;
esac
;;
esac
exit 0

```

样例脚本将尝试用两个时刻来重启本地 Web 服务：

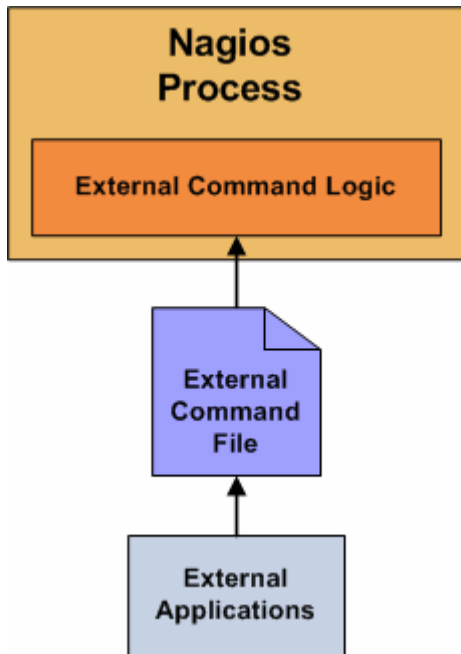
- 在服务检测出三次并且是处于软态紧急状态之后；
- 在服务首次进入硬态紧急状态之后；

这个脚本理论上在服务转入硬态故障之前可以重启 HTTP 服务并可以修复故障，这里包含了首次重启没有成功的情况。须注意的是事件处理将只是第一次进入硬态紧急状态时才会执行事件处理，这将阻止 Nagios 在服务一直处于硬态故障的状态时会反复不停地重启动 Web 服务。你不需要反复地重启，对吧？ :-)

这就是事件处理。事件处理很容易理解、编写和实现，所以要尽量尝试来使用并看看它能给你带来什么。

6.9. 外部命令

6.9.1. 介绍



Nagios 可以处理并执行外部应用包括 CGI 程序并给出按其监控时所得到的运行结果给出报警。外部应用可以在 [命令文件](#) 中给定，它可以被 Nagios 守护程序定期地处理并执行。

6.9.2. 使能外部命令

为使 Nagios 可以处理外部命令，必须按如下步骤来做：

- 使能了外部检测命令 [check_external_commands](#) 选项。
- 用 [command_check_interval](#) 选项设置了命令检测的频度。
- 在 [command_file](#) 选项中指定了命令文件的位置。
- 对包含有外部命令文件的目录给出了恰当的目录操作权限，象在 [快速指南](#) 说明的那样。

6.9.3. Nagios 什么时候用外部命令检测？

- 由 [command_check_interval](#) 选项指定了一个规格化的频度，该选项在主配置文件中给出。
- 在 [事件处理句柄](#) 之后被立即执行。在规格化定制周期执行的命令检测之后增加了如果需要 Nagios 来做事件处理之后立即执行的要求。

6.9.4. 使用外部命令

外部命令可以完善各种在 Nagios 运行中需要做的事情。例如临时性地对某些服务或主机的报警不做响应，临时取消对服务的检测，强制对服务进行检测，增加对主机或服务的批注等等。

6.9.5. 命令格式

外部命令可以写入到 [命令文件](#) 之中，用如下格式：

```
[time] command_id;command_arguments
```

这里的 **time** 是指用 **time_t** 格式的时间戳，标记外部命令或应用执行时间。而 **command_id** 的值和 **command_arguments** 命令参数取决于 Nagios 将执行的命令。

一个完整的外部命令列表包括如何使用这些的样例可以在线查阅 URL：

<http://www.nagios.org/developerinfo/externalcommands/>

6.10. 状态类型

6.10.1. 介绍

被监控的主机和服务的当前状态由如下两个要素决定：

- 主机与服务的状况(如正常、警告、运行和宕机等)
- 服务与主机将要从属的状态类型

Nagios有两种状态类型 — 软态和硬态。这两种状态取决于监控逻辑，当执行过 [事件处理](#)或是当 [通知](#)被初始送出时将会给出决定。

本文试图描述软态和硬态的状态区别，它们是如何发生及在发生时将做些什么。

6.10.2. 服务与主机的检测重试

为防止因瞬态故障而引发错误报警，Nagios 需要定义主机与服务经过多少次的重试检测后再认为故障是“真正”发生。这个次数是由主机与服务中的 **max_check_attempts** 选项决定。理解如果真正故障发生时主机与服务进行检测重试的做法在理解状态类型机制很重要。

6.10.3. 软态

软态在如下情况时会发生：

- 当服务与主机检测返回一个非正常或非运行状态，同时，服务与主机的重试检测还没有达到设置 **max_check_attempts** 所设定的次数时。这个被称为软故障。
- 当服务与主机自软故障转变时。这个被称为软恢复。

软态变化时将有如下情形发生：

- 软态被记录；
- 事件处理将会执行以捕获分析软态；

只是在使能了主配置文件里的 [log_service_retries](#)选项或是 [log_host_retries](#)选项时软态才会被记录。

真正重要的是在软态发生时去执行事件处理。在它转入硬态之前应用事件处理将特别有效，如果你试图预处理修复故障时。当事件处理运行时，宏 [\\$HOSTSTATETYPE\\$](#)或 [\\$SERVICESTATETYPE\\$](#)将会赋值“软态”，这样将使事件处理脚本得知什么时候正确动作。更多有关事件处理的信息可以查阅 [这篇文档](#)。

6.10.4. 硬态

主机与服务的硬态将会在如下情况发生：

- 当服务与主机检测返回一个非正常或非运行状态，同时，服务与主机的重试检测已经达到设置 **max_check_attempts** 所设定的次数时；这个被称为硬故障。
- 当主机与服务从一个硬故障转变为另一个时(如告警到紧急)；
- 当服务检测处理非正常状态时对应的主机处于宕机或不可达时；
- 当主机或服务自一个硬态恢复时；这个被称为硬恢复。
- 当收到一个 [强制主机检测](#)结果时。强制主机检测结果将被认定是硬态除非设置使能了 [passive host checks are soft](#)选项；

当主机或服务经过硬态变迁时如下情形将会发生：

- 硬态被记录；
- 事件处理将会执行以处置硬态；
- 在主机与服务故障和恢复时对应的联系人将收到通知；

当执行事件处理时宏 [\\$HOSTSTATETYPE\\$](#)或 [\\$SERVICESTATETYPE\\$](#)将会赋值为“硬态”，这样将使事件处理脚本得知什么时候正确动作。更多有关事件处理的信息可以查阅 [这篇文档](#)。

6.10.5. 举例

这里有一个在当状态转换发生时和当事件处理与通知被送出时如何给定状态类型的例子。服务的最大重试次数 **max_check_attempts** 值设置为 3。

表 6.4.

时刻	检测次数	状态	状态类型	是否状态变换	注释
0	1	正常	硬态	否	初始的服务状态
1	1	紧急	软态	是	首次发现非正常状态。执行事件处理。

时刻	检测次数	状态	状态类型	是否状态变换	注释
2	2	告警	软态	是	服务仍处于非正常状态。执行事件处理。
3	3	紧急	硬态	是	达到最大重试次数，服务状态类型进入硬态。事件处理执行且送出故障通知。检测数在当时被重置为 1。
4	1	告警	硬态	是	服务状态变换为硬态告警。事件处理执行且送出故障通知。
5	1	告警	硬态	否	服务仍停在硬态故障，为个取决于服务的通知间隔是多少，也可能会有另一个故障通知被送出。
6	1	正常	硬态	是	服务经历了一个硬态恢复。事件处理执行且一个恢复通知被送出。
7	1	正常	硬态	否	服务仍处于正常。
8	1	未知	软态	是	服务被检查出从一个软态非正常态变换了。事件处理执行。
9	2	正常	软态	是	服务经历了一个软恢复。事件处理执行，但通知不会送出，因为这不是个“真正”故障。当这发生时状态类型设置为硬态而且检测次数被立即重置为 1。
10	1	正常	硬态	否	服务停在了一个正常状态。

6.11. 主机检测

6.11.1. 介绍

这里将介绍主机检测的基本机制...

6.11.2. 什么时候做主机检测？

由 Nagios 守护进程来做主机检测，一般是：

- 在规格化的间隔内，这个由 [主机对象定义](#)里的`check_interval`和`retry_interval`选项确定；
- 当主机状态变换后对应的服务做按需检测；
- 在 [主机可达性](#)逻辑中需要做按需检测；
- 在 [主机依赖检测的前处理](#)中需要做按需检测；

规格化定期主机检测是可选的，如果你将主机对象定义里的 `check_interval` 值设置为 0，Nagios 将不会定期做检测。然而它仍旧会在按需检测时做主机检测，如果由监控逻辑中的其他部分需要进行检测时。

按需检测被用于当绑定于某台主机上的服务状态变换时对主机检测，因为 Nagios 需要知道主机是否有状态变换情况发生。服务状态的变化通常表征着主机状态也发生变化。例如，如果 Nagios 发现某台主机上的 HTTP 服务从“紧急”到“正常”时，它也有也表示主机刚刚从重启中恢复它重新恢复运行。

按需检测同样被用于 [主机可达性](#)逻辑之中对主机检测。Nagios被设计为尽快地得到网络概况，且尽快分辨出主机的宕机与不可达状态。这些完全不同的状态将协助管理员尽快在网络中定位出问题源点。

按需检测同样在 [主机依赖性检测的前处理](#)逻辑中进行主机检测。这将协助确保得到尽可能正确的依赖逻辑关系。

6.11.3. 缓存主机检测

可用缓存检测来显著地改善按需检测的性能，缓存检测机制可使Nagios放弃一个主机的检测执行而使用相关的最近检测来替代，更多有关缓存检测的信息可查阅 [这篇文档](#)。

6.11.4. 依赖性与检测

可通过给出对象定义里的 [主机依赖](#)定义来防止Nagios因对一个主机状态的检测而对一个或更多主机进行状态检测。更多的关于主机依赖关系的信息可查阅 [这篇文档](#)。

6.11.5. 并发主机检测

计划式主机检测是并发运行的。当 Nagios 要运行一个计划的主机检测时，初始会对它进行主机检测返回后再然后做其他工作(运行服务检测等)。一个主机检测程序是由主 Nagios 守护进程 fork 派生出来的一个子进程。当主机检测完成，子进程将通告主进程检测的结果。Nagios 主进程将处理检测结果并采取合适的动作(执行事件处理、发送通知等)。

如果需要按需主机检测同样可以并发。在前面所提及的，Nagios 如果可以利用从缓存的相关的最近主机检测的结果而放弃一次按需检测。

当Nagios处理计划的和按需的主机检测结果时，它可能初始化之后的其他主机检测。初始化这些检测可能是由于两个原因：[依赖性检测的前处理](#)和使用 [网络可达性](#)逻辑来判定主机状态。初始化的之后检测一般是并发的。然而，一个很大问题必须要把握，这将降低运行效率...



注意

在主机对象定义里将`max_check_attempts`值设定为 1 会导致一系列性能问题。原因就是，如果Nagios需要使用 [网络可达性](#)逻辑来判定一个主机的真正状态(它们是宕机或不可达)时，Nagios将不得不对该主机的直接父节点执行一连续地检测。需要重申，这些检测是一个个地连续运行，而不是并发，这将导致性能降低。基于此，建议总是将主机对象定义里的`max_check_attempts`域值设置大于 1。

6.11.6. 主机状态

主机在如下三种状态之一时会被检测：

- 运行 (UP)
- 宕机 (DOWN)
- 不可达 (UNREACHABLE)

6.11.7. 主机状态判定

主机检测由 [插件](#)来做，插件会返回结果，结果是运行、告警、未知和紧急四个状态之一。那么Nagios将如何把插件的返回值转换成主机的运行、宕机或不可达呢？下面会讲到。

下表给出了插件返回结果与预置主机状态，之后会做某些后续处理(后面会讲到)，后续处理可能会改变最终的主机状态。

表 6.5. 状态值

插件结果	预置主机状态
正常 (OK)	运行 (UP)
告警 (WARNING)	运行 (UP) 或宕机 (DOWN) *
未知 (UNKNOWN)	宕机 (DOWN)
紧急 (CRITICAL)	宕机 (DOWN)

注意



告警通常意味着主机是运行的，然而，如果你使能了 [use aggressive host checking](#)选项的话，告警也可理解为主机宕机。

如果预置主机状态是宕机，Nagios 将尝试它是否真的宕机还是它是不可达。宕机与不可达分开很重要，这使得管理员更快地查找到网络故障的源头。下面给出了基于该主机的父节点得出主机最终状态的表格。主机的父节点是在对象定义里的 `parents` 域来设定的。

表 6.6.

预置主机状态	父节点状态	最终的主机状态
宕机 (DOWN)	至少一台运行 (UP)	宕机 (DOWN)
宕机 (DOWN)	全部父节点不是宕机 (DOWN) 就是不可达 (UNREACHABLE)	不可达 (UNREACHABLE)

有关如何分辨宕机 (DOWN) 与不可达 (UNREACHABLE) 状态的更多信息可查阅 [这篇](#) 文档。

6.11.8. 主机状态变换

你可能注意到了主机并不总是留在一种状态，事件中断、打上补丁和服务需要重新启动等都会让它状态变换。当Nagios检测出主机状态时，它总是要感知到主机从四种状态之间做了变换并要采取对应的行动。这些在不同的 [状态类型](#) (硬态或软态) 下的状态变换将会触发 [事件处理](#) 的运行和发送出 [通知](#)。发现与处置这些状态变换是Nagios该做的全部。

当主机状态过度频繁地变换状态时可以考虑状态处于“抖动” (flapping)。一个明显的例子就是一台主机由于加载操作系统而不断地重启动，这种状态就是处于抖动。不得不应对它是个有趣的方案，Nagios能感知主机开始抖动，并且可以压制通知直到抖动停下来达到一种稳定状态。更多的有关感知抖动逻辑的内容可以查阅 [这篇](#) 文档。

6.12. 服务检测

6.12.1. 介绍

下面将对服务检测的基本机制进行说明...

6.12.2. 什么时候会做服务检测？

由 Nagios 守护进行的服务检测执行于

- 在规划的间隔到了时；间隔由 [服务对象定义](#) 里的 `check_interval` 和 `retry_interval` 选项确定。
- 因 [服务依赖检测的前处理](#) 需要而发出的按需检测；

因 [服务依赖检测的前处理](#) 逻辑而做的按需检测可以保证得到的依赖逻辑关系尽可能准确。如果不使用 [使用依赖](#)，Nagios将不做任何按需服务检测。

6.12.3. 缓存服务检测

通过应用缓存服务检测可以显著地改善按需服务检测的性能，缓存服务检测可令Nagios放弃一个服务检测而用一个相关的最近一个检测来替代。如果给出了 [服务依赖](#)，缓存检测将只是提高性能。更多的有关缓存检测可查阅 [这篇](#) 文档。

6.12.4. 依赖性与检测

通过给出 [服务依赖](#) 对象的定义可防止 Nagios 为判定一个服务而对一个或多个服务进行状态检测。更多的有关依赖检测的信息可查阅 [这篇](#) 文档。

6.12.5. 服务检测并发

计划的服务检测是并发运行。当 Nagios 需要运行一个计划服务检测时，它将初始化一个服务检测并返回来做其他工作（运行主机检测等）。服务检测在一个由 Nagios 守护主进程中派生出的子进程中运行，子进程将把检测结果通告给主进程。Nagios 主程序会处理检测结果并采取合适的行动（执行一个事件处理、发出通知等）。

如果需要，按需服务检测同样可以并发。如前所述，Nagios 可以放弃一个按需检测如果可以利用缓存的最近的检测结果来替代的话。

6.12.6. 服务状态

被检测的服务有下列四种状态之一：

- 正常 (OK)
- 告警 (WARNING)
- 未知 (UNKNOWN)
- 紧急 (CRITICAL)

6.12.7. 服务状态判定

由 [插件](#) 来做的服务检测将返回一个状态，是正常 (OK)、告警 (WARNING)、未知 (UNKNOWN) 或紧急 (CRITICAL) 四种之一。插件直接将转换为服务状态，如插件返回一个告警状态将使一个服务处于告警态。

6.12.8. 服务状态变换

当 Nagios 对服务进行状态检测，将会感知到服务在四种状态之间进行变化并采取合适行动。这些状态有不同的 [状态类型](#) (硬态或软态) 将会触发 [事件处理](#) 运行和发出 [通知](#)。服务状态变换同样可以触发按需的 [主机检测](#)。感知与处理状态变换是 Nagios 该做的全部。

当服务状态过分频繁地变换可被认为处于“抖动”。Nagios 可以感知到服务开始抖动，可压制通知直到抖动结束并且服务达到某种稳定态。更多的关于感知抖动逻辑的信息可以查阅 [这篇](#) 文档。

6.13. 自主检测

6.13.1. 介绍

Nagios 用两种模式来对主机和服务进行检测：自主检测和强制检测。强制检测将在 [其他地方](#) 说明，这里只涉及自主检测。自主检测是最通用的监控主机与服务的方式。自主检测的主要特点是：

- 由 Nagios 进程进行起始的自主检测
- 自主检测是在一个规格化预定义周期之上进行

6.13.2. 自主检测是如何进行的？

自主检测由 Nagios 守护进程的检测逻辑进程初始化。当 Nagios 需要进行对主机和服务进行状态检测时，它将需要检测的信息传给一个插件，由插件来检测主机或服务并给出一个可供进一步运作的状态，将结果返给 Nagios 守护进程。Nagios 按照主机或服务的结果来做适当地动作（如发出告警、执行事件处理句柄等）

有关插件是如何工作的更多信息可以在 [这里](#)找到。

6.13.3. 什么时间执行自主检测？

自主检测将在如下情况执行：

- 当规格化时间到达时；规格化时间由主机和服务定义的 `check_interval` 和 `retry_interval` 选项决定。
- 进程必须处于守护状态；

规格化计划检测发生的间隔要么是`check_interval`要么是`retry_interval`，这取决于主机与服务当前处于什么 [状态类型](#)。如果主机与服务是处于硬态，实际检测间隔将等于`check_interval`值，如果它处于软态，检测间隔将等于`retry_interval`值。

每当Nagios需要取得某特定主机或服务的最新状态时，将会去做按需检测。例如当Nagios要判断主机的 [可达性](#)时，它通常会去做针对主机父节点及子节点的按需检测以决定该网段的状态。按需检测同样发生于 [依赖性检测的前处理](#)逻辑之中，以确保Nagios得到最为准确的状态信息。

6.14. 强制检测

6.14.1. 介绍

通常情况下Nagios监控主机与服务使用规格化计划表来做 [自主检测](#)。自主检测使用“轮询”机制来对设备或服务状态信息进行收集，这是常见方式。Nagios同样支持用另一种方式，即强制方式来替代自主方式来检测，强制检测的关键特性是：

- 强制检测被外部应用或进程初始化和执行；
- 强制检测的结果交给 Nagios 来处理；

自主检测与强制检测的最主要不同是自主检测是由 Nagios 来做初始化和执行而强制检测是由外部应用程序来做。

6.14.2. 强制检测的用处

强制检测在如下监控中很有用：

- 本身是异步的并且无法有效地基于一个规格化计划表来轮询的监控；
- 被监控主机位于防火墙后面无法从监控服务器送出自主检测；

异步式服务的例子是自身提供包括 SNMP 陷阱或安全警告等强制监控方式的服务。从来不会知道在一个指定时间片段里将会收到多少 SNMP 陷阱或安全警告，所以这些不适合用每几分钟来判定一下被监控的状态。

强制检测也可以用于配置一个 [分布式](#) 监控或是一个 [冗余](#) 监控系统。

6.14.3. 强制检测是如何工作的？

更详细的强制检测的工作机制是...

- 一个外部应用对主机或服务的状态进行检查；
- 外部程序将检测结果写入 [外部命令文件](#) 之中；
- 每次 Nagios 读入外部命令文件并将全部强制检测结果写入一个将要处理的队列中，该队列同样会保存自主检测结果；
- Nagios 将定期执行 [检测结果接收的事件处理](#) 并扫描结果队列。在队列里可找到的每个服务检测结果都会同样处理 – 不管这个检测结果是自主检测的还是强制检测的结果 – Nagios 将按照检测结果送出通知、记录警告等。

对自主检测与强制检测的处理本质上是一致的，这使得 Nagios 与其他的外部应用无缝集成。

6.14.4. 使能强制检测

在 Nagios 里使能强制检测需要做如下设置：

- 将 [accept_passive_service_checks](#) 域设置为 1；
- 在主机与服务对象定义里将 `passive_checks_enabled` 域设定为 1；

如果想全局地关闭强制检测，将 [accept_passive_service_checks](#) 域设置为 0；

如果只想对几个主机与服务关闭强制检测，在对象与服务对象定义里用 `passive_checks_enabled` 域来控制。

6.14.5. 提交服务的强制检测结果

外部应用通过写入一个 `PROCESS_SERVICE_CHECK_RESULT` [外部命令](#) 到外部命令文件中来告诉 Nagios 提交了一个强制检测结果。

命令的格式是：

```
[<timestamp>]
PROCESS_SERVICE_CHECK_RESULT;<host_name>;<svc_description>;<return_code>;<plugin_output>
```

参数说明：

- **timestamp** 是一个 `time_t` 格式的时间戳来表征检测动作的时间，注意有在方括号的右侧有一个空格；

- **host_name** 是主机与服务对象定义里的短名称;
- **svc_description** 是指定服务对象定义里的服务描述;
- **return_code** 是返回的检测结果 (0=正常 (OK), 1=报警 (WARNING), 2=紧急 (CRITICAL), 3=未知 (UNKNOWN));
- **plugin_output** 是服务检测的文本输出 (如同插件输出)。

注意



必须在 Nagios 提交服务对象定义后才可以提交检测结果; Nagios 将会忽略没有最后一次启动后读入的配置文件里所做对象定义的全部检测结果。

提示



一个用SHELL脚本来实现强制检测并将结果提交给Nagios的例子可以在文档 [可变服务](#)里找到。

6.14.6. 提交主机的强制检测结果

外部应用通过写一个 `PROCESS_HOST_CHECK_RESULT` 外部命令到外部命令文件中来告诉 Nagios 提交了一个强制检测结果。

命令格式是:

```
[<timestamp>].PROCESS_HOST_CHECK_RESULT;<host_name>;<host_status>;<plugin_output>
```

参数说明:

- **timestamp** 是一个 `time_t` 格式的时间戳来表征检测动作的时间, 注意有在方括号的右侧有一个空格;
- **host_name** 是主机对象定义里的短名称;
- **host_status** 是主机的状态 (0=运行 (UP), 1=宕机 (DOWN), 2=不可达 (UNREACHABLE));
- **plugin_output** 是服务检测的文本输出 (如同插件输出)。



必须在 Nagios 提交主机对象定义后才可以提交检测结果; Nagios 将会忽略没有最后一次启动后读入的配置文件里所做对象定义的全部检测结果。

6.14.7. 强制检测与主机状态

与自主检测不同, Nagios (默认) 不会在强制检测时尝试判定主机是宕机 (DOWN) 或不可达 (UNREACHABLE)。Nagios 把强制检测结果当做真实的主机状态, 并且不会使用 [网络可达性检测逻辑](#) 来判定

主机的真正状态。如果是想对远程主机的强制检测进行判定时将会导致问题，同样，在一个 [分布式监控](#) 环境下因父/子节点的关系不一样时也会有问题。

可以设置令Nagios在强制检测的状态是宕机 (DOWN) / 不可达 (UNREACHABLE) 时变换到一个“合理”的状态，通过设置 [translate passive host checks](#) 变量来做变换即可，更详细地关于如何设置它的信息可以查阅 [这篇](#) 文档。

注意



强制主机检测一般认定是 [硬态](#) 类型，除非使能了 [passive host checks are soft](#) 选项时才会不同。

6.14.8. 判定来自远程主机的强制检测结果

如果发送主机与服务强制检测结果的外部应用与 Nagios 同属一台主机，那么外部应用可以很容易地象上面所说的那样直接将结果写入外部命令文件，然而，当应用程序在远程主机上时这样做并不容易。

为了让远程主机可以发送强制检测结果到安装有Nagios的监控服务器上，我开发了名为 [NSCA](#) 外部构件。NSCA外部构件包括一个服务守护进程运行在装有Nagios的主机上，另一个客户端安装于远程主机上。服务守护进程将监听来自远程客户端的联接，对来自远程的结果做些基本的确认，然后将结果直接写入外部命令文件之中(象上面所描述的那样)。更多的关于NSCA外部构件的信息可以查阅 [这篇](#) 文档。

第 7 章 运行 Nagios 的基本操作

第 7 章 运行 Nagios 的基本操作

7.1. 验证配置文件的正确性

每次修改过你的 [配置文件](#)，你应该运行一次检测程序来验证配置的正确性。在运行你的Nagios程序之前这是很重要的，否则的话会导致Nagios服务因配置的错误而关闭。

为验证你配置，运行 Nagios 带命令行参数 `-v`，象这样：

```
/usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
```

如果你确实忘记了一些重要的数据或是错误地配置了，Nagios 将会给出一个报警或是一个错误信息，其中会给出错误的位置。错误信息通常会打印出错误配置的文件中的那一行。在错误时，Nagios 通常是在预检查出有问题打印出问题的源配置文件行后退回到命令行状态。这使得 Nagios 不会因一个错误而落入需要验证一个因错误而嵌套的配置循环错误之中。报警信息可**通常**是被忽略的，因为一般那些只是建议性的并非必须的。

一旦你已经验证了你配置文件并修改过你的错误，就可以继续下去，[启动或重启Nagios](#) 服务了。

7.2. 启动与停止 Nagios

有多于一种方式来启动、停止和重新启动 Nagios，这里在有更通常做的方式...

提示



在你启动或重新启动你的Nagios程序之前，你总是要确保你 [验证你的配置文件](#) 已经通过。

7.2.1. 启动 Nagios

- 初始化脚本：最简单的启动 Nagios 守护进程的方式是使用初始化脚本，象这样：

```
/etc/rc.d/init.d/nagios start
```

- 手工方式：你可以手动地启动 Nagios 守护进程，用命令参数-d，象这样：

```
/usr/local/nagios/bin/nagios -d /usr/local/nagios/etc/nagios.cfg
```

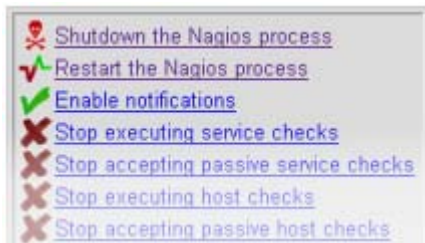
7.2.2. 重新启动 Nagios

当你修改了配置文件并想使之生效的话，重新启动或重载入动作是必须的。

- 初始化脚本：最简单地重新启动 Nagios 守护进程的方式是使用初始化脚本，象这样：

```
/etc/rc.d/init.d/nagios reload
```

- Web 接口方式：你可以利用 WEB 接口，通过点击“进程信息”的超链接页面里的“重新启动 Nagios 进程”来重新启动 Nagios，见图



- 手工方式：你可以手动地发一个 SIGHUP 信号，象这样：

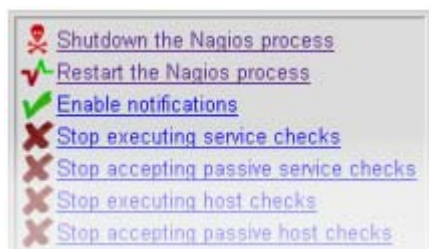
```
kill -HUP <nagios_pid>
```

7.2.3. 停止 Nagios

- 初始化脚本：最简单地停止 Nagios 守护进程的方式是通过初始化脚本，象这样：

```
/etc/rc.d/init.d/nagios stop
```

- Web 接口方式：你可以利用 WEB 接口，通过点击“进程信息”的超链接页面里的“关闭 Nagios 进程”来停止 Nagios，见图



- 手工方式：你可以手动发一个 SIGTERM 信号，象这样：

```
kill <nagios_pid>
```

7.3. 快速启动选项

7.3.1. 介绍

只有很少几件事可以减少 Nagios 的启动或重启总时间。加速启动方法包括有移除些负担还包括加快配置文件处理过程。

利用这些技术在如下一种或几种情况时特别有效：

- 大型安装配置
- 复杂地配置(过度地利用模板特性)
- 需要进行频繁重启的安装模式

7.3.2. 背景

每次 Nagios 启动和重启时，在它着手进行监控工作之前必须要处理配置文件。启动过程中的配置处理包括如下几步：

- 读入配置文件
- 解析模板定义
- 重粘连(“Recombobulating”)对象(是我想到的应做各种工作)
- 复制对象定义
- 继承对象属性
- 对象定义排序
- 验证对象关联关系的完整性
- 验证回路
- 和其他...

当有很大的或是很复杂的配置文件要处理时有几步非常消耗时间的。有没有加快这些的办法？当然有！

7.3.3. 评估启动时间

在做让启动速度更快的事情之前，需要看看可能性有多少和是否有必要涉足此事。这个比较容易——只是用 `-s` 命令行开关启动 Nagios 以取得计时和调度信息。

下面是个输出样例(做过精减，只是显示了有关部分)，在这个例子中，假定 Nagios 配置为对 25 个主机和超过 10,000 个服务进行监控。

```
/usr/local/nagios/bin/nagios -s /usr/local/nagios/etc/nagios.cfg
Nagios 3.0-prealpha
Copyright (c) 1999-2007 Ethan Galstad (http://www.nagios.org)
Last Modified: 01-27-2007
License: GPL

Timing information on object configuration processing is listed
below. You can use this information to see if precaching your
object configuration would be useful.

Object Config Source: Config files (uncached)

OBJECT CONFIG PROCESSING TIMES      (* = Potential for precache savings with -u option)
-----
Read:                               0.486780 sec
Resolve:                            0.004106 sec *
Recomb Contactgroups: 0.000077 sec *
Recomb Hostgroups:    0.000172 sec *
Dup Services:         0.028801 sec *
Recomb Servicegroups: 0.010358 sec *
Duplicate:             5.666932 sec *
Inherit:               0.003770 sec *
Recomb Contacts:      0.030085 sec *
Sort:                 2.648863 sec *
Register:             2.654628 sec
Free:                 0.021347 sec
=====
```



```
TOTAL:                11.555925 sec  * = 8.393170 sec (72.63%) estimated savings
```

Timing information on configuration verification is listed below.

```
CONFIG VERIFICATION TIMES          (* = Potential for speedup with -x option)
```

```
-----  
Object Relationships: 1.400807 sec
```

```
Circular Paths:        54.676622 sec  *
```

```
Misc:                  0.006924 sec  
=====
```

```
TOTAL:                56.084353 sec  * = 54.676622 sec (97.5%) estimated savings
```

OK, 看看发生了什么。先看汇总信息, 大概有 11.6 秒用于处理配置文件有 56 秒来验证配置。这意味着每次用这个配置启动或重启 Nagios 时, 它大约会有 68 秒来做启动事项而不会做任何监控的事情! 如果是在定制配置 Nagios 过程中也是不可容忍的。

那么怎么办? 看一下输出内容, 如果运用了优化选项, Nagios 将可以在配置读取过程节省大约 8.4 秒而在验证过程可节省 63 秒。

哇! 从 68 秒到只有 5 秒?! 是的! 看看下面是怎么做到的。

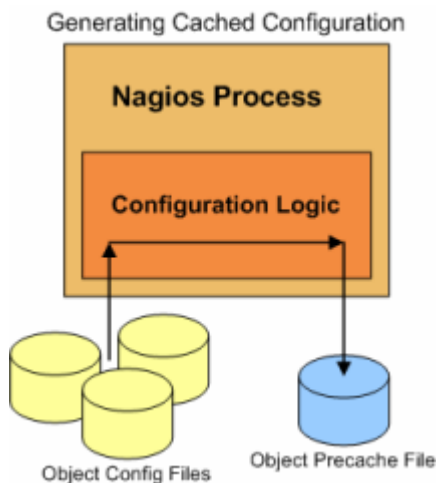
7.3.4. 预缓存对象配置

Nagios 可在解析配置文件过程中做些加速, 特别是当配置中使用了模板来做继承等的时候。为降低 Nagios 解析配置文件的处理时间可用 Nagios 预处理与预缓存配置文件的函数。

当用 -p 命令参数来运行 Nagios 时, Nagios 将读入配置文件, 处理后将配置结果写入预缓存文件(由主配置文件中 [precached_object_file](#) 域指定文件位置)。该预缓存配置文件将包含了预处理后的信息将使 Nagios 处理配置文件更容易和快捷。必须把 -p 参数选项与 -v 或 -s 命令参数一起使用, 如下例。注意要做预缓存配置文件之前配置应是已被验证过的。

```
/usr/local/nagios/bin/nagios -pv /usr/local/nagios/etc/nagios.cfg
```

预缓存配置文件有大小明显地比原有配置文件大。这是正常的由设计初衷决定的。



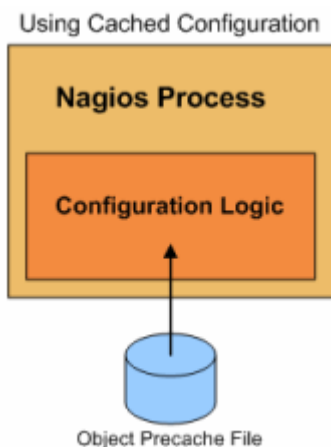
一旦预缓存对象配置文件创建，可以启动 Nagios 时带上 `-u` 命令行选项以让它使用预缓存配置文件而不是配置文件本身。

```
/usr/local/nagios/bin/nagios -ud /usr/local/nagios/etc/nagios.cfg
```

重要



如果更改了配置文件，必须在 Nagios 重新启动前要重新验证和重建预缓存配置文件。如果没有重建预缓存配置文件，Nagios 将使用旧配置运行因为是由旧配置生成的预缓存文件，而不是用新的原始配置文件。



7.3.5. 跳过回路检测

第二步(也是最耗时)部分是对配置中的回路进行检测。在上面例子中这一步几乎用去了 1 分钟来验证配置验证。

什么时回路检测和为什么要做这么长时间？回路检测逻辑是为了确保在你的主机、主机依赖、服务和依赖服务等对象之间不存在任何的循环路径。如果在配置中有循环路径，Nagios 将会因死锁而停止。用时较长原因是由于没有使用较高效的算法。欢迎提供更高效发现回路的算法。提示：这意味着 EMail 给我有关 Nagios 论文的计算机科学系研究生将有机会得到些回赠代码。:-)

如果你想在 Nagios 为启动时跳过回路检测，可以在命令行回加上 `-x` 参数，象这样：

```
/usr/local/nagios/bin/nagios -xd /usr/local/nagios/etc/nagios.cfg
```

重要



当要在启动和重启前跳过回路检测之前，验证配置文件的正确性是非常非常重要的！没有这么做将有可能导致 Nagios 逻辑上的死锁。你已被我提醒过了啊！

7.3.6. 联合起来使用

按照下面步骤将会使用预缓存配置文件并且跳过回路检测以充分加速启动。

1、验证配置文件并生成预缓存配置文件，用如下命令：

```
/usr/local/nagios/bin/nagios -vp /usr/local/nagios/etc/nagios.cfg
```

2、如果 Nagios 正在运行，停掉它；

3、启动 Nagios，让其使用预缓存配置文件而且跳过回路检测：

```
/usr/local/nagios/bin/nagios -uxd /usr/local/nagios/etc/nagios.cfg
```

4、当更改了原始配置文件时，需要重新启动 Nagios 并修改现有内容，重新回到步骤 1 去验证配置并重构预缓存配置文件。一旦做好了，就可以通过 Web 接口来重启 Nagios 或是在系统中发个 SIGHUP 信号，如果没有重构预缓存配置文件，Nagios 将用旧配置运行，因为它首先会读入缓存配置文件而不是源配置文件；

5、就这么多！祝你可以加快启动过程。

7.4. 关于 CGI 程序模块的信息

7.4.1. 说明

这里将描述一下随 Nagios 发行的几个 CGI 程序模块，每个 CGI 模块都需要做充分的授权设置。默认情况下 CGI 程序将依赖于你在 Web 服务程序里的授权和对你所请求的视图给你的授权。更多的有关授权配置的信息可以在 [这里](#) 找到。

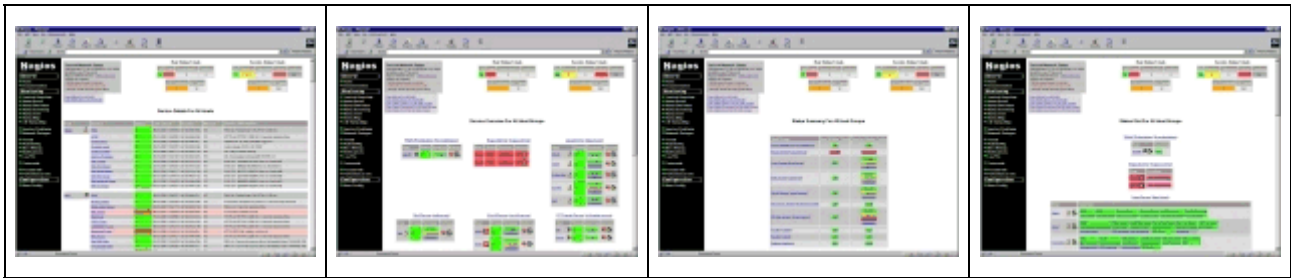
7.4.2. 索引

- [Status CGI](#)
- [Status map CGI](#)
- [WAP interface CGI](#)
- [Status world CGI \(VRML\)](#)
- [Tactical overview CGI](#)
- [Network outages CGI](#)
- [Configuration CGI](#)

- [Command CGI](#)
- [Extended information CGI](#)
- [Event log CGI](#)
- [Alert history CGI](#)
- [Notifications CGI](#)
- [Trends CGI](#)
- [Availability reporting CGI](#)
- [Alert histogram CGI](#)
- [Alert summary CGI](#)

Status CGI

表 7.1.



模块文件名 *status.cgi*

描述: 在 Nagios 里这是一个很重要的 CGI 模块。它可以让你观测到被监测的全部主机和服务的当前状态。它将生成本个主机类型的输出报告 — 全部的（或部分主机）以成组方式给出状态报告和全部的服务（或部分主机上的全部服务）的状态。

授权要求:

- 如果你已被 授权对全部主机 你就可以看到全部主机和全部服务。
- 如果你已被 授权对全部的服务 你就可以看到全部服务。
- 如果你是一个被 授权的联系人 你就可以看到以你为联系人的全部主机和服务。

Status Map CGI



模块文件名 *statusmap.cgi*

描述: 这个CGI模块将创建一个基于你监测网络全部主机的二维地图。使用Thomas Boutell的 [gd](#)库(版本是 1.6.3 或更高)来生成一个PNG图,里面的二维坐标依赖于每个 [主机](#)对象的定义(包括可以给每个主机定义一个好看的图标)。如果你宁可让CGI程序自己自动地设定主机的坐标,用一下这个 [default_statusmap_layout](#)域来指定一个二维图生成算法。

授权要求:

- 如果你已被 **授权对全部主机**你就可以看到全部主机。
- 如果你是一个被**授权的联系人**你就可以看到以你为联系人的主机。

注意

没有被授权的用户只能看到那些主机的节点处于**未知**状态。我真的让它无法看到**任何东西**,如果你无法看到主机依赖的话,你甚至无法看到一个二维图...

WAP Interface CGI



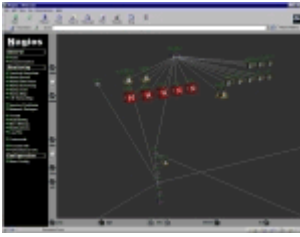
模块文件名 *statuswml.cgi*

描述: 这个 CGI 模块将给 WAP 接口提供网络状态服务。如果你有一个 WAP 设备(象一个带因特网接入能力的移动电话),你可以在移动中观看状态信息。在主机组汇总、主机概览、主机详细信息、服务详细信息、全部的故障告警、全部未处理故障等等不同的报告,除了状态信息外,同样可以从移动电话里来设置取消告警、关闭检测和通知故障等。这个功能很酷吧?

授权要求:

- 如果你已被 **授权看系统信息**你可以看到Nagios进程信息。
- 如果你已被 **授权对全部主机**你可以看到全部主机和服务的状态数据。
- 如果你已被 **授权对全部的服务**你可以看到全部服务的状态数据。
- 如果你是一个被**授权的联系人**你可以看到以你做为联系人的主机和服务的状态数据。

Status World CGI (VRML)



模块文件名 *statuswrl.cgi*

描述: 这个CGI模块将对你所监控网络的全部主机生成一个三维虚拟视图。这些绘制中所用的主机三维坐标(以及渲染图片)来自于配置文件中的 [主机](#) 定义。如果你想让CGI程序模块自动地生成三维坐标, 可以设置 [default_statuswrl_layout](#) 域来指定一个三维图坐标生成算法。同样, 在你要做观察之前你也应在你系统里安装一个虚拟现实的浏览器(象 [Cortona](#)、[Cosmo Player](#) 或 [WorldView](#))。

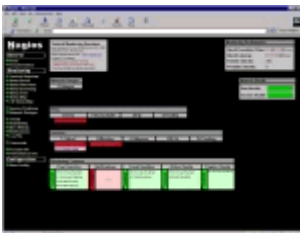
授权要求:

- 如果你已被 **授权对全部主机** 你就可以看到全部主机。
- 如果你是一个被**授权的联系人** 你就可以看到以你为联系人的主机。

注意

对于没有被授权的用户, 将在没授权的主机节点上看到**未知**状态。我真的让他无法看到**任何东西**, 如果你无法看到主机依赖关系时你甚至无法看到一个三维图...

Tactical Overview CGI



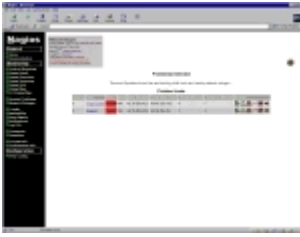
模块文件名 *tac.cgi*

描述: 这个 CGI 模块给了一个网络活动的“鸟瞰图”。这容许你快速地得到网络概况、主机状态和服务状态。在已被“处理”的故障(象被认同的和关闭告警的故障)和没有被捕获的问题之间做出区分辨别, 且是需要提请关注的。如果你在监控大量的主机和服务并且想只是用一组画面来分析处理这些故障的话这个会很有用。

授权要求:

- 如果你已被 **授权对全部主机** 你可以看到全部主机和全部服务。
- 如果你已被 **授权对全部的服务** 你可以看到全部服务。
- 如果你是一个被**授权的联系人** 你就可以看到以你为联系人的全部主机和服务。

Network Outages CGI



模块文件名 *outages.cgi*

描述：这个 CGI 将给出你网络中的引发网络出错的“问题”主机列表。这对于管理一个大型的网络和想快速定位网络故障来源的情况是很有用的。列表中的主机将按出错问题的先后关系来排列。

授权要求：

- 如果你已被 **授权对全部主机** 你就可以看到全部主机。
- 如果你是一个被**授权的联系人** 你就可以看到以你为联系人的主机。

Configuration CGI



模块文件名 *config.cgi*

描述：这个 CGI 模块将让你可以看到全部对象 (象主机、主机组、联系人、联系人组、时间周期、服务等) 的配置，这些配置写在你的 [对象配置文件](#) 里面。

授权要求：

- 你必须被 **授权可以看到任何配置** 信息和任意一种配置内容。

Command CGI



模块文件名 *cmd.cgi*

描述：这个 CGI 模块将让你给 Nagios 进程发出命令。虽然它有很多个命令参数，但你最好是独立地使用它们。在不同的 Nagios 版本间它们有很大地不同。用 [extended information CGI](#) 模块来做为发布命令的起点。

授权要求：

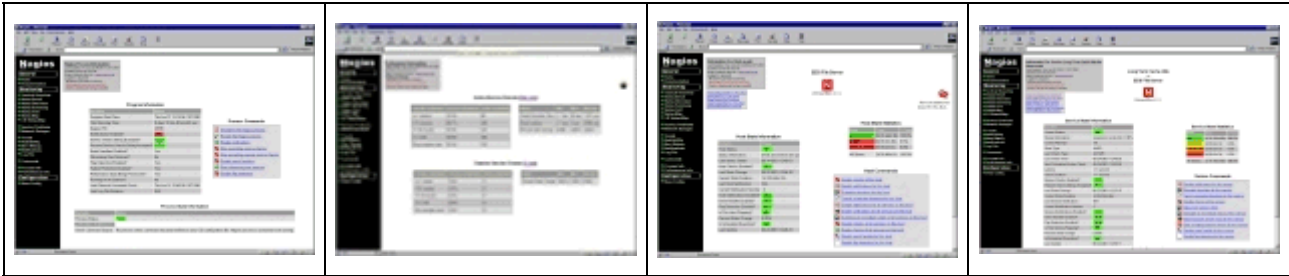
- 你必须被 授权做系统命令 以使你发出对Nagios有影响的命令(重启动、关闭、模式切换等等)。
- 如果你被 授权在全部主机上执行命令 以使你可以对全部主机和服务发出命令。
- 如果你被 授权对全部服务执行命令 以使你可对全部服务发出命令。
- 如果你是一个被授权的联系人 你可以对你做为联系人的主机和服务上发出命令。

注意

如果没有使用在CGI配置文件里 [use_authentication](#) 选项，这个CGI模块将不会让你对Nagios执行任何命令，这是对你设置的一种保护。如果你决定在WEB里设置成非授权管理状态来运行，我建议你最好移走这个CGI模块。

Extended Information CGI

表 7.2.



模块文件名 *extinfo.cgi*

描述：这个CGI模块将让你看到Nagios进程信息、主机和服务状态统计、主机和服务注释和其他信息等。同样它也可以做为对Nagios发出命令的服务，跟 [command CGI](#) 模块一样。虽然它有几个命令参数，但你最好是独立地用它们 — 在不同的Nagios版本之间它们会有不同。你可以通过点击在页面边上的“网络健康状况”和“进程信息”里的链接来进到这个CGI模块，也可以通过点击 [status CGI](#) 里的主机或服务上的链接进入。

授权要求：

- 你必须被 授权看系统信息 以使你可以看到进程信息报告。
- 如果你已被 授权对全部主机 你可以看到全部主机和服务的扩展信息。
- 如果你已被 授权对全部的服务 你可以看到全部服务的扩展信息。
- 如果你是一个被授权的联系人 你可以看到以你做联系人的全部主机与服务的扩展信息。

Event Log CGI



模块文件名 *showlog.cgi*

描述：此CGI模块用于显示 [日志文件](#)。如果已设置 [日志回滚](#) 使能，可以用顶部的导航链接来在打包的日志文件中浏览当前告警。

授权要求：

- 你必须被 [授权看系统信息](#) 以使你可看到日志文件报告。

Alert History CGI



模块文件名 *history.cgi*

描述：这个CGI模块被用于显示部分或是全部主机的历史故障。这个是显示 [日志文件CGI模块](#) 信息的子集。你可以过滤显示输出内容，只挑出指定类型的故障来查看(如按硬故障和软故障分类，或按服务和主机告警的类型来显示等)。如果你设置了 [日志回滚](#)，你可以通过页面顶端的导航链接来在打包的日志文件中查看当前的历史信息。

授权要求：

- 如果你已被 [授权对全部主机](#) 你可以看到全部主机和服务的历史信息。
- 如果你已被 [授权对全部的服务](#) 你可以看到全部服务的历史信息。
- 如果你是一个被[授权的联系人](#) 你可以看到以你做为联系人的全部服务和主机的历史信息。

Notifications CGI



模块文件名 *notifications.cgi*

描述: 这个CGI模块可以用于显示给各类联系人而发出主机和服务的通知。这个输出是 The output is basically a subset of the information that is displayed by the [日志CGI模块](#)显示内容的子集。你可以过滤输出显示内容，只是显示指定的通知类型(如服务通知、主机通知、给指定联系人的通知等)。如果设置了 [日志回滚](#)选项使能，你可以通过在页面顶端的导航链接来在打包的日志文件中查看当前的通知。

授权要求:

- 如果你已被 [授权对全部主机](#)你可以查看全部的主机和服务的通知报告。
- 如果你已被 [授权对全部的服务](#)你可以查看全部服务的通知。
- 如果你是一个被[授权的联系人](#)你可以查看以你为联系人的全部服务和主机的通知报告。

Trends CGI



模块文件名 *trends.cgi*

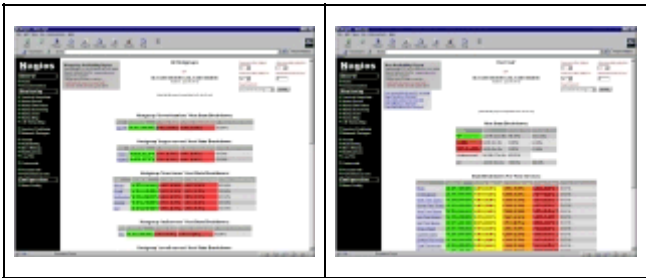
描述: 这个CGI模块可以创建一个主机或服务的任意时间段内的状态趋势图。为了让此CGI模块更有用，你需要设置 [日志回滚](#)选项使能并保留好打包的日志文件，打包日志文件保留路径在 [log_archive_path](#)域里设置。这个CGI模块使用了Thomas Boutell的 [gd](#)库(版本 1.6.3 或更高)以创建状态趋势图。

授权要求:

- 如果你已被 [授权对全部主机](#)你可以查看全部主机和全部服务的趋势图
- 如果你已被 [授权对全部的服务](#)你可以查看全部服务的趋势图。
- 如果你是一个被[授权的联系人](#)你可以查看以你为联系人的全部服务和主机的趋势图。

Availability Reporting CGI

表 7.3.



模块文件名 *avail.cgi*

描述: 这个CGI模块可用于查看用户定制的指定时间段内的可用性报告。为使这个CGI程序更多地被运用,你要设置 [日志回滚](#)使能并保留打包的日志文件,日志文件保存于 [log_archive_path](#)域里面。

授权要求:

- 如果你已被 [授权对全部主机](#)你可以查看全部主机和全部服务的可用性数据报告。
- 如果你已被 [授权对全部的服务](#)你可以查看全部服务的可用性数据报告。
- 如果你是一个被[授权的联系人](#)你可以查看以你为联系人的全部服务和主机的可用性数据报告。

Alert Histogram CGI



模块文件名 *histogram.cgi*

描述: 这个CGI模块可用于显示在用户定制的时间段内的主机和服务的可用性曲线。为使这个CGI更多地利用,你须设置 [日志回滚](#)选项并保留你的打包日志文件,日志文件保存于 [log_archive_path](#)域设置的路径里。这个CGI模块使用了Thomas Boutell的 [gd](#)库(版本 1.6.3 或更高)以创建历史曲线图。

授权要求:

- 如果你已被 [授权对全部主机](#)你可以查看全部的主机和全部服务的历史曲线。
- 如果你已被 [授权对全部的服务](#)你可以查看全部服务的历史曲线。
- 如果你是一个被[授权的联系人](#)你可以查看以你为联系人的全部服务和主机的历史曲线报告。

Alert Summary CGI



模块文件名 *summary.cgi*

描述: 这个 CGI 模块提供了有关主机和服务告警的概要性的报告,包括总的和最大的告警源等。

授权要求:

- 如果你已被 [授权对全部主机](#)你可以查看全部主机和全部服务的汇总信息。
- 如果你已被 [授权对全部的服务](#)你可以查看全部服务的汇总信息。
- 如果你是一个被[授权的联系人](#)你可以查看以你为联系人的全部服务和主机的汇总信息。

第 8 章 Nagios 深入进阶

第 8 章 Nagios 深入进阶

8.1. Nagios 的插件

8.1.1. 介绍

与其他的监控工具不同，Nagios 的内在机制中不包含针对主机和服务状态的检测，而是依赖于外部程序（称为插件）来做这些脏活（——真正该做的检查工作是脏活，真够幽默的）。

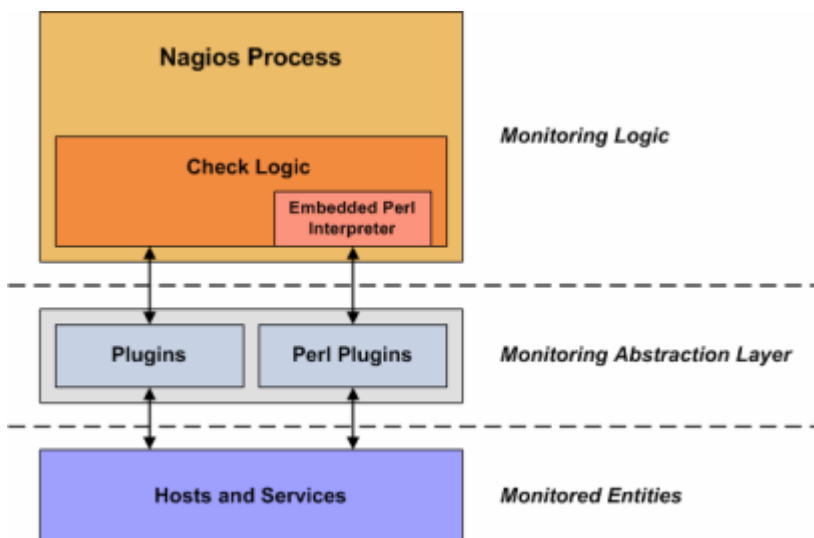
8.1.2. 什么是插件？

插件是编译的执行文件或脚本（Perl 脚本、SHELL 脚本等等），可以在命令行下执行对主机或服务状态检查。Nagios 运行这些插件的检测结果来决定网络中的主机和服务的当前状态。

当需要检测主机或服务状态时 Nagios 总是执行一个插件程序，插件总要做点**事情**（注意一般条件下）来完成检查并给出简洁的结果给 Nagios。Nagios 将处理这些来自插件的结果并做些该做的动作（运行 [事件处理句柄](#)、发送出 [告警](#)等）。

8.1.3. 插件是一个抽象层

插件扮演了位于 Nagios 守护程序里的监控逻辑和实际被监控的主机与服务之间的抽象层次。



在插件构架之上你可以监控所有你想要监控的东西。如果你能自动地处理检测过程你就可以用 Nagios 来监控它。已经写好很多插件以用于监控基础性资源象处理器负荷、磁盘利用率、PING 包率等，如果你想监控点别的，你需要查阅 [书写插件](#) 这篇文档并自己付出努力，这很简单地！

在插件构架之下，事实上 Nagios 也不知道你想要搞些什么名堂。你可以监控网络流量态势、数据错包率、房间温度、CPU 电压值、风扇转速、处理器负载、磁盘空间或是有可能在早上起来你的超级无敌的

面包机烤出正宗的色泽... Nagios 不会理解什么被监控了一它只是忠实地记录下了这些被管理资源的状态变化轨迹。只有插件自己知道监控了什么东西并如何完成检测。

8.1.4. 什么样的插件可用？

有许多插件可用于监控不同的设备和服务，包括：

- HTTP、POP3、IMAP、FTP、SSH、DHCP
- CPU 负荷、磁盘利用率、内存占用、当前用户数
- Unix/Linux、Windows 和 Netware 服务器
- 路由器和交换机
- 等等

8.1.5. 获得插件

插件不与 Nagios 包一起发布，但你可以下载到 Nagios 官方插件和由 Nagios 用户书写并维护的额外插件，在这些网址里：

- Nagios Plugins工程 <http://nagiosplug.sourceforge.net/>
- Nagios下载页面 <http://www.nagios.org/download/>
- NagiosExchange.org<http://www.nagiosexchange.org/>

8.1.6. 如何来使用插件 X？

当你在命令行下用命令参数-h 或-help 运行时许多插件会显示基本用法信息。例如如果你想知道如何使用 check_http 插件或是它的可接收哪些选项参数时，你只要尝试运行：

```
./check_http --help
```

就可以看到提示内容了。

8.1.7. 插件 API

你可以在 [这里](#)找到有关插件技术论述的信息并且有如何书写你自己定制插件的内容。

8.2. 理解 Nagios 宏及其工作机制

8.2.1. 宏

Nagios 是如此地柔性化的一个重要特征是具备在命令域的定义里使用宏。宏允许你的命令里获取主机、服务和其他对象源的信息。

8.2.2. 宏替换 — 宏的工作机制

在 Nagios 执行命令之前，它将对命令里的每个宏替换成它们应当取得的值。这种宏替换发生在 Nagios 在执行各种类型的宏时候 — 象主机和服务的检测、通知、事件处理等。

有些特定的宏包含了其他宏，这些宏包括\$HOSTNOTES\$、\$HOSTNOTESURL\$、\$HOSTACTIONURL\$、\$SERVICENOTES\$、\$SERVICENOTESURL\$和\$SERVICEACTIONURL\$。

8.2.3. 例 1：主机 IP 地址宏

当在命令定义中使用主机或服务宏时，宏将要执行所用的值是指向主机或服务所带有值。尝试这个例子，假定在 `check_ping` 命令定义里使用了一个主机对象，象这样：

```
define host{
    host_name          linuxbox
    address            192.168.1.2
    check_command      check_ping
    ...
}

define command{
    command_name      check_ping
    command_line      /usr/local/nagios/libexec/check_ping -H $HOSTADDRESS$ -w
100.0,90% -c 200.0,60%
}
```

那么执行这个主机检测命令时展开并最终执行的将是这样的：

```
/usr/local/nagios/libexec/check_ping -H 192.168.1.2 -w 100.0,90% -c 200.0,60%
```

很简单，对吧？优美之处在于你可以在只用一个命令定义来完成无限制的多个主机的检测。每个主机可以使用相同的命令来进行检测，而在对他们检测之前将把主机地址正确地替换。

8.2.4. 例 2：命令参数宏

同样你可以向命令传递参数，这样可以保证你的命令定义更具通用性。参数指定在对象（象主机或服务）中定义，用一个“!”来分隔他们，象这样：

```
define service{
    host_name          linuxbox
    service_description PING
    check_command      check_ping!200.0,80%!400.0,40%
    ...
}
```

在上例中，服务的检测命令中含有两个参数(请参考 [\\$ARGn\\$](#)宏)，而\$ARG1\$宏将是“200.0,80%”，同时\$ARG2\$将是“400.0,40%”(都不带引号)。假定使用之前的主机定义并这样来定义你的check_ping命令：

```
define command{  
    command_name    check_ping  
    command_line    /usr/local/nagios/libexec/check_ping -H $HOSTADDRESS$ -w  
$ARG1$ -c $ARG2$  
}
```

那么对于服务的检测命令最终将是这样子的：

```
/usr/local/nagios/libexec/check_ping -H 192.168.1.2 -w 200.0,80% -c 400.0,40%
```

提示



如果你需要在你的命令行里使用这个(!)字符，你得加上转义符反斜线(\)，就是你要写成(\!)。如果想用反斜线，同样得加转义符，写成(\\)。

8.2.5. 按需而成的宏(on-demand macro)

通常在命令对象定义里使用主机和服务的宏，用以在命令执行时指向某个服务或是主机。但也就是说，一个在对命名为linuxbox的主机上执行命令时，全部的 [标准的主机宏](#)都应使用这个主机值都是正运行的主机名linuxbox。

如果不想这样，也是是让命令里引用的主机或服务宏指向另外一些主机或服务，你可以用“按需生成的宏”的机制。除了那个需要指定从哪个给主机或服务时取值而包含在内的标识之外，按需而成的宏看起来就象是一般的宏。这里是基本的“按需而成的宏”的基本格式：

- \$HOSTMACRONAME:host_name\$
- \$SERVICEMACRONAME:host_name:service_description\$

用标准的主机和服务的宏名字替换HOSTMACRONAME和SERVICEMACRONAME，这些标准的宏可以在 [这里](#)查到。

要注意宏的名字与主机和服务的标识之间隔有一个(:)符号。为了形成表达按需而成的服务宏的标识，在标识里既有主机名又有服务描述—他们俩用一个(:)符号分开。

提示



按需而成的服务宏可以包含主机名域为空，此时所绑定的主机由服务结合情况自行来指定。

下面是按需而成的主机和服务宏的例子：

```
$HOSTDOWNTIME:myhost$                <--- On-demand host macro
$SERVICESTATEID:novellserver:DS Database$  <--- On-demand service macro
$SERVICESTATEID::CPU Load$            <--- On-demand service macro
with blank host name field
```

按需而成的宏同样可以运用于主机组、服务组、联系人和联系人组宏里，例如：

```
$CONTACTEMAIL:john$                  <--- On-demand contact macro
$CONTACTGROUPMEMBERS:linux-admins$    <--- On-demand contactgroup
macro
$HOSTGROUPALIAS:linux-servers$        <--- On-demand hostgroup macro
$SERVICEGROUPALIAS:DNS-Cluster$      <--- On-demand servicegroup
macro
```

8.2.6. 用户自定义宏

在主机、服务或联系人等对象里的任何一个 [用户自定义变量](#) 都可以联接宏。用户自定义的变量宏命名如下：

- `$_HOSTvarname$`
- `$_SERVICEvarname$`
- `$_CONTACTvarname$`

如下的主机对象定义中定义了一个用户自定义变量是“_MACADDRESS”，见细节：

```
define host{
    host_name          linuxbox
    address             192.168.1.1
    _MACADDRESS         00:01:02:03:04:05
    ...
}
```

那么主机对象的 _MACADDRESS 用户自定义变量的值就可以在宏 `$_HOSTMACADDRESS$` 里面使用。你可以在 [这里](#) 找到更多的关于用户自定义变量以及如何在宏里使用它的信息。

8.2.7. 宏的清理

在命令执行之前，有些宏要去掉那些可能会引起SHELL潜在风险的元字符。这些元字符由

[illegal_macro_output_chars](#)选项来定义。下面这些宏是要做这种处理的：

- [\\$HOSTOUTPUT\\$](#)
- [\\$LONGHOSTOUTPUT\\$](#)
- [\\$HOSTPERFDATA\\$](#)
- [\\$HOSTACKAUTHOR\\$](#)
- [\\$HOSTACKCOMMENT\\$](#)
- [\\$SERVICEOUTPUT\\$](#)
- [\\$LONGSERVICEOUTPUT\\$](#)
- [\\$SERVICEPERFDATA\\$](#)
- [\\$SERVICEACKAUTHOR\\$](#)
- [\\$SERVICEACKCOMMENT\\$](#)

8.2.8. 作为环境变量的宏

由Nagios将宏变成一个操作系统的环境变量将有利于在脚本或命令执行时引用。为保证安全和清晰的思路，[\\$USERn\\$](#)和“按需而成on-demand”的主机和服务宏是不可以被作为环境变量的。

环境变量的命名与其包含的命名标准宏(列表在[这里](#))的名字是相关的，它们的名字前面加前缀“NAGIOS_”。比如说 [\\$HOSTNAME\\$](#)宏在环境变量里被命名为“NAGIOS_HOSTNAME”。

8.2.9. 可用宏

所有的在Nagios里的可用的宏以及如何使用它们的列表可以在[这里](#)查找。

8.3. Nagiosr 内嵌的标准宏

这里列出了Nagios里可用的标准宏。按需生成的宏和用户定制变量宏在[这篇文档](#)里有说明。

8.3.1. 宏的有效性

虽然宏可被用于定义的各种命令之中，但并非每种宏在特定环境里是“合法”的。如，有些宏只是在服务通知命令里有效，而另外一些只在主机检测命令里有用。Nagios 可以辨识和处理的情况有十种不同类型，它们就是：

- 服务检测
- 服务通知
- 主机检测
- 主机通知
- 服务 [事件处理](#)和全局服务事件处理
- 主机 [事件处理](#)和全局主机事件处理

- [OCSP](#)命令
- [OCHP](#)命令
- 服务 [性能数据](#)命令
- 主机 [性能数据](#)命令

下面表格中列出了在 Nagios 可用的全部的宏，并且每个宏都有一个简短说明及什么样命令是有效的。如果宏在无效的命令中使用，可能会被空串替代。须注意全部宏是大写字符且名字里最前和最后都有\$字符。

8.3.2. 可利用的宏图表

表 8.1. 图例:

No	该宏不可用
Yes	该宏可以运用

表 8.2. 主机宏: ³

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$HOSTNAME\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTDISPLAYNAME\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTALIAS\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTADDRESS\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTSTATE\$	Yes	Yes	Yes ¹	Yes	Yes	Yes	Yes	Yes
\$HOSTSTATEID\$	Yes	Yes	Yes ¹	Yes	Yes	Yes	Yes	Yes
\$LASTHOSTSTATE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$LASTHOSTSTATEID\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTSTATETYPE\$	Yes	Yes	Yes ¹	Yes	Yes	Yes	Yes	Yes
\$HOSTATTEMPT\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$MAXHOSTATTEMPTS\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTEVENTID\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$LASTHOSTEVENTID\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTPROBLEMID\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$LASTHOSTPROBLEMID\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTLATENCY\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTEXECUTIONTIME\$	Yes	Yes	Yes ¹	Yes	Yes	Yes	Yes	Yes
\$HOSTDURATION\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTDURATIONSEC\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTDOWNTIME\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTPERCENTCHANGE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTGROUPNAME\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTGROUPNAMES\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$LASTHOSTCHECK\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$LASTHOSTSTATECHANGE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$LASTHOSTUP\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$LASTHOSTDOWN\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$LASTHOSTUNREACHABLE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTOUTPUT\$	Yes	Yes	Yes ¹	Yes	Yes	Yes	Yes	Yes
\$LONGHOSTOUTPUT\$	Yes	Yes	Yes ¹	Yes	Yes	Yes	Yes	Yes
\$HOSTPERFDATA\$	Yes	Yes	Yes ¹	Yes	Yes	Yes	Yes	Yes

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$HOSTCHECKCOMMAND\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTACKAUTHOR\$ ⁸	No	No	No	Yes	No	No	No	No
\$HOSTACKAUTHORNAME\$ ⁸	No	No	No	Yes	No	No	No	No
\$HOSTACKAUTHORALIAS\$ ⁸	No	No	No	Yes	No	No	No	No
\$HOSTACKCOMMENT\$ ⁸	No	No	No	Yes	No	No	No	No
\$HOSTACTIONURL\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTNOTESURL\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTNOTES\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$TOTALHOSTSERVICES\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$TOTALHOSTSERVICESOK\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$TOTALHOSTSERVICESWARNING\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$TOTALHOSTSERVICESUNKNOWN\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$TOTALHOSTSERVICESCRITICAL\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

表 8.3. 主机组宏：

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$HOSTGROUPALIAS\$ ⁵	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTGROUPMEMBERS\$ ⁵	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTGROUPNOTES\$ ⁵	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTGROUPNOTESURL\$ ⁵	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTGROUPACTIONURL\$ ⁵	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

表 8.4. 服务宏：

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理 与 OCSP	主机事件处理 与 OCHP	服务性能	主机性能
\$SERVICEDESC\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEDISPLAYNAME\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICESTATE\$	Yes ²	Yes	No	No	Yes	No	Yes	No
\$SERVICESTATEID\$	Yes ²	Yes	No	No	Yes	No	Yes	No
\$LASTSERVICESTATE\$	Yes	Yes	No	No	Yes	No	Yes	No
\$LASTSERVICESTATEID\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICESTATETYPE\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEATTEMPT\$	Yes	Yes	No	No	Yes	No	Yes	No
\$MAXSERVICEATTEMPTS\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEISVOLATILE\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEEVENTID\$	Yes	Yes	No	No	Yes	No	Yes	No
\$LASTSERVICEEVENTID\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEPROBLEMID\$	Yes	Yes	No	No	Yes	No	Yes	No
\$LASTSERVICEPROBLEMID\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICELATENCY\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEEXECUTIONTIME\$	Yes ²	Yes	No	No	Yes	No	Yes	No
\$SERVICEDURATION\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEDURATIONSEC\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEDOWNTIME\$	Yes	Yes	No	No	Yes	No	Yes	No

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理 与 OCSP	主机事件处理 与 OCHP	服务性能	主机性能
\$SERVICEPERCENTCHANGE\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEGROUPNAME\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEGROUPNAMES\$	Yes	Yes	No	No	Yes	No	Yes	No
\$LASTSERVICECHECK\$	Yes	Yes	No	No	Yes	No	Yes	No
\$LASTSERVICESTATECHANGE\$	Yes	Yes	No	No	Yes	No	Yes	No
\$LASTSERVICEOK\$	Yes	Yes	No	No	Yes	No	Yes	No
\$LASTSERVICEWARNING\$	Yes	Yes	No	No	Yes	No	Yes	No
\$LASTSERVICEUNKNOWN\$	Yes	Yes	No	No	Yes	No	Yes	No
\$LASTSERVICECRITICAL\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEOUTPUT\$	Yes ²	Yes	No	No	Yes	No	Yes	No
\$LONGSERVICEOUTPUT\$	Yes ²	Yes	No	No	Yes	No	Yes	No
\$SERVICEPERFDATA\$	Yes ²	Yes	No	No	Yes	No	Yes	No
\$SERVICECHECKCOMMAND\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICEACKAUTHOR\$ ⁸	No	Yes	No	No	No	No	No	No
\$SERVICEACKAUTHORNAME\$ ⁸	No	Yes	No	No	No	No	No	No
\$SERVICEACKAUTHORALIAS\$ ⁸	No	Yes	No	No	No	No	No	No
\$SERVICEACKCOMMENT\$ ⁸	No	Yes	No	No	No	No	No	No
\$SERVICEACTIONURL\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICENOTESURL\$	Yes	Yes	No	No	Yes	No	Yes	No
\$SERVICENOTES\$	Yes	Yes	No	No	Yes	No	Yes	No

表 8.5. 服务组宏：

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$SERVICEGROUPALIAS ⁶	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$SERVICEGROUPMEMBERS ⁶	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$SERVICEGROUPNOTES ⁶	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$SERVICEGROUPNOTESURL ⁶	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$SERVICEGROUPACTIONURL ⁶	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

表 8.6. 联系人宏：

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$CONTACTNAME\$	No	Yes	No	Yes	No	No	No	No
\$CONTACTALIAS\$	No	Yes	No	Yes	No	No	No	No
\$CONTACTEMAIL\$	No	Yes	No	Yes	No	No	No	No
\$CONTACTPAGER\$	No	Yes	No	Yes	No	No	No	No
\$CONTACTADDRESSn\$	No	Yes	No	Yes	No	No	No	No

表 8.7. 联系人组宏：

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$CONTACTGROUPALIAS ⁷	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$CONTACTGROUPMEMBERS ⁷	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

表 8.8. 汇总统计宏：

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$TOTALHOSTSUP\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALHOSTSDOWN\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALHOSTSUNREACHABLE\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALHOSTSDOWNUNHANDLED\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALHOSTSUNREACHABLEUNHANDLED\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALHOSTPROBLEMS\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALHOSTPROBLEMSUNHANDLED\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALSERVICESOK\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALSERVICESWARNING\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALSERVICESCRITICAL\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALSERVICESUNKNOWN\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALSERVICESWARNINGUNHANDLED\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALSERVICESCRITICALUNHANDLED\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALSERVICESUNKNOWNUNHANDLED\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALSERVICEPROBLEMS\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes
\$TOTALSERVICEPROBLEMSUNHANDLED\$ ¹⁰	Yes	Yes ⁴	Yes	Yes ⁴	Yes	Yes	Yes	Yes

表 8.9. 通知宏:

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$NOTIFICATIONTYPE\$	No	Yes	No	Yes	No	No	No	No
\$NOTIFICATIONRECIPIENTS\$	No	Yes	No	Yes	No	No	No	No

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$NOTIFICATIONISESCALATED\$	No	Yes	No	Yes	No	No	No	No
\$NOTIFICATIONAUTHOR\$	No	Yes	No	Yes	No	No	No	No
\$NOTIFICATIONAUTHORNAME\$	No	Yes	No	Yes	No	No	No	No
\$NOTIFICATIONAUTHORALIAS\$	No	Yes	No	Yes	No	No	No	No
\$NOTIFICATIONCOMMENT\$	No	Yes	No	Yes	No	No	No	No
\$HOSTNOTIFICATIONNUMBER\$	No	Yes	No	Yes	No	No	No	No
\$HOSTNOTIFICATIONID\$	No	Yes	No	Yes	No	No	No	No
\$SERVICENOTIFICATIONNUMBER\$	No	Yes	No	Yes	No	No	No	No
\$SERVICENOTIFICATIONID\$	No	Yes	No	Yes	No	No	No	No

表 8.10. 日期/时间宏:

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$LONGDATETIME\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$SHORTDATETIME\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$DATE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$TIME\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$TIMET\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$ISVALIDTIME:\$ ⁹	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$NEXTVALIDTIME:\$ ⁹	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

表 8.11. 文件宏:

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$MAINCONFIGFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$STATUSDATAFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$COMMENTDATAFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes ⁵
\$DOWNTIMEDATAFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$RETENTIONDATAFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$OBJECTCACHEFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$TEMPFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$TEMPPATH\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$LOGFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$RESOURCEFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$COMMANDFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$HOSTPERFDATAFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$SERVICEPERFDATAFILE\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

表 8.12. 其他宏：

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$PROCESSSTARTTIME\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$EVENTSTARTTIME\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$ADMINEMAIL\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$ADMINPAGER\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
\$ARGn\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

宏名	服务检测	服务通知	主机检测	主机通知	服务事件处理与 OCSP	主机事件处理与 OCHP	服务性能	主机性能
\$USERn\$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

8.3.3. 宏的描述说明

表 8.13. 主机宏：³

\$HOSTNAME\$	主机简称(如“biglinuxbox”), 取自于 主机定义 里的host_name域。
\$HOSTDISPLAYNAME\$	可供替代显示的主机名, 取自于 主机定义 里的display_name域。
\$HOSTALIAS\$	主机全称、匿名或是描述, 取自于 主机定义 里的alias域。
\$HOSTADDRESS\$	主机地址。取自于 主机定义 里的address域。
\$HOSTSTATE\$	当前主机状态的说明字符串(“运行”、“宕机”或“不可达”)。
\$HOSTSTATEID\$	当前主机状态的标识数字(0=运行、1=宕机、2=不可达)。
\$LASTHOSTSTATE\$	最后主机状态的说明字符串(“运行”, “宕机”或“不可达”)。
\$LASTHOSTSTATEID\$	最后主机状态的标识数字(0=运行、1=宕机、2=不可达)。
\$HOSTSTATETYPE\$	主机检测时指示主机当前 状态类型 的字符串(“硬态”或“软态”)。软态是指当主机检测返回一个非正常状态并且开始进行重试时所处状态的状态类型。硬态是指当主机检测已经达到最大检测次数后所处的状态的状态类型。
\$HOSTATTEMPT\$	主机检测当前的重试次数。比如, 如果第二次要进行重检测, 该宏的值是2。当前尝试次数只是反应出当主机事件处理处于软态时基于重试次数内执行指定动作的重试次数。
\$MAXHOSTATTEMPTS\$	最大重试次数由当前主机对象定义给出。当写入软态时的主机事件处理做指定动作的重试时将会用到。
\$HOSTEVENTID\$	全局的唯一 ID 值, 指示当前主机状态, 每次主机或服务经历一次状态变换, 全局的事件 ID 计数器增 1。如果主机没有经历状态变换, 该值将置

	为 0。
\$LASTHOSTEVENTID\$	给定主机的前一个(全局唯一的)事件 ID 值。
\$HOSTPROBLEMID\$	全局分配的主机当前故障状态的唯一标识值。每次主机(或服务)自一个运行(UP)或正常(OK)状态变换到故障状态时,全局故障 ID 值会增 1。如果主机当前是非运行状态该宏将是一个非零值。主机状态在两个非运行状态(如宕机到不可达)之间变换时将不会导致全局故障 ID 值增 1。如果主机当前处于运行状态,该宏将被置 0。与事件处理相结合,该宏将被用于当主机首次进入故障状态时系统自动地打开一个事故操作票。
\$LASTHOSTPROBLEMID\$	对指定主机的前一次全局唯一故障 ID 值。与事件处理相结合,该宏将用于当主机恢复到运行状态时自动地关闭一个事故操作票。
\$HOSTLATENCY\$	一个浮点数值的秒数,该值记录了 预期的主机检测 迟后于它计划检测时间的秒数。比如,如果计划检测时间是 03:14:15 但直到另一时刻 03:14:17 才执行时,这个数据将是 2.0 秒。按需地主机检测固定只有一个 0 秒的迟后。
\$HOSTEXECUTIONTIME\$	一个浮点数值的秒数,该值记录了主机花了多少秒来执行主机检测程序。
\$HOSTDURATION\$	字符串表示的主机当前状态共保持了多长时间,格式是“XXh YYm ZZs”指出了时、分和秒。
\$HOSTDURATIONSEC\$	主机当前状态共保持了多少秒的数值。
\$HOSTDOWNTIME\$	当前主机“停机时间”的秒值。如果当前主机处于一个 计划停机时间 之内,该值将大于 0,如果主机不处于停机时间内,这个值是 0。
\$HOSTPERCENTCHANGE\$	主机所经历的状态变化率(一个浮点数值)。该值被用于 抖动检测 算法。
\$HOSTGROUPNAME\$	主机所从属的主机组的短名称。这个值取自 主机组对象定义 里的 hostgroup_name 域。如果该主机从属于一个或多个主机组时该宏将只包含其中的一个(应该是所从属的第一个主机组的组名,按源程序是这样理解——译者注)。
\$HOSTGROUPNAMES\$	主机所从属的全部主机组名的列表,多个主机组名时它是个用逗号分隔的

	列表。
\$LASTHOSTCHECK\$	一个 time_t (UNIX 系统的秒计数器) 格式的时间戳指向主机最后一次检测的执行时间。
\$LASTHOSTSTATECHANGE\$	一个 time_t (UNIX 系统的秒计数器) 格式的时间戳指向主机最后一次状态改变的时间。
\$LASTHOSTUP\$	一个 time_t (UNIX 系统的秒计数器) 格式的时间戳指向主机检测出仍处于运行 (UP) 状态的最后时间。
\$LASTHOSTDOWN\$	一个 time_t (UNIX 系统的秒计数器) 格式的时间戳指向主机检测出仍处于宕机 (DOWN) 状态的最后时间。
\$LASTHOSTUNREACHABLE\$	一个 time_t (UNIX 系统的秒计数器) 格式的时间戳指向主机仍处于不可达 (UNREACHABLE) 状态的最后时间。
\$HOSTOUTPUT\$	最后一次主机检测输出的首行内容 (如 "Ping OK")。
\$LONGHOSTOUTPUT\$	最后一次主机检测输出的除首行之外的内容。
\$HOSTPERFDATA\$	最后一次主机检测所返回的全部 性能数据 。
\$HOSTCHECKCOMMAND\$	用于主机检测的带有全部参数的命令串。
\$HOSTACKAUTHOR\$ ⁸	用字符串表示的用户名, 该用户是主机问题的确认者。该宏只是在通知动作里的 \$NOTIFICATIONTYPE\$ 宏是 "ACKNOWLEDGEMENT" 时才是合法有效的。
\$HOSTACKAUTHORNAME\$ ⁸	字符串表示的联系人 (如果刚好有的话) 短名字, 该联系人是主机问题的确认者。该宏只是在通知动作里的 \$NOTIFICATIONTYPE\$ 宏是 "ACKNOWLEDGEMENT" 时才是合法有效的。
\$HOSTACKAUTHORALIAS\$ ⁸	字符串表示的联系人 (如果刚好有的话) 昵称, 该宏只是在通知动作里的 \$NOTIFICATIONTYPE\$ 宏是 "ACKNOWLEDGEMENT" 时才是合法有效的。
\$HOSTACKCOMMENT\$ ⁸	字符串表示的确认注释字符串, 这是由用户对主机问题做出确认时录入的。该宏只是在通知动作里的 \$NOTIFICATIONTYPE\$ 宏是 "ACKNOWLEDGEMENT" 时才是合法有效的。

\$HOSTACTIONURL\$	主机动作的 URL 串，该宏可能包含另一些宏(如\$HOSTNAME\$)，当需要向 Web 页传入主机名时这个宏很有用。
\$HOSTNOTESURL\$	主机注释 URL 串，该宏可能包含另一些宏(如\$HOSTNAME\$)，当需要向 Web 页传入主机名时这个宏很有用。
\$HOSTNOTES\$	主机的注释。该宏可能包含另一些宏(如\$HOSTNAME\$)，当需要给主机指定一些特定信息时很有用，如在主机的描述里。
\$TOTALHOSTSERVICES\$	主机被绑定的服务总数。
\$TOTALHOSTSERVICESOK\$	主机被绑定的服务处于正常(OK)状态的数量。
\$TOTALHOSTSERVICESWARNING\$	主机被绑定的服务处于告警(WARNING)状态的数量。
\$TOTALHOSTSERVICESUNKNOWN\$	主机被绑定的服务处于未知(UNKNOWN)状态的数量。
\$TOTALHOSTSERVICESCRITICAL\$	主机被绑定的服务处于紧急(CRITICAL)状态的数量。

表 8.14. 主机组宏：⁵

\$HOSTGROUPALIAS\$ ⁵	有两种情况：要么是主机组名被认为是给按需宏参数的主机组的长名称或昵称，要么是当前主机所从属的首要主机组(如果上下文里不是用于一个按需宏的话)的长名称或昵称。这个值取自 主机组对象定义 里的 <code>alias</code> 域。
\$HOSTGROUPMEMBERS\$ ⁵	一个用逗号分隔的全部主机列表，它有两种情况：要么是被认为是给按需宏当参数的主机列表，要么是当前主机所从属的首要主机组(如果上下文里不是用于一个按需宏的话)的全部主机列表。
\$HOSTGROUPNOTES\$ ⁵	是一个注释，有两种情况：要么是给按需宏当参数的主机组名，要么是当前主机所从属的首要从属的主机组(如果上下文里不是用于一个按需宏的话)所含有的注释。它取自 主机组对象定义 里的 <code>notes</code> 域。
\$HOSTGROUPNOTESURL\$ ⁵	The notes URL associated with either 1) the hostgroup name passed as an on-demand macro argument or 2) the primary hostgroup associated with the current host (if not used in the context of an on-demand macro). 它取自 主机组对象定义 里的 <code>notes_url</code> 域。

<code>\$HOSTGROUPNOTES⁵</code>	The action URL associated with either 1) the hostgroup name passed as an on-demand macro argument or 2) the primary hostgroup associated with the current host (if not used in the context of an on-demand macro). 它取自 主机组对象定义 里的 action_url 域。
---	--

表 8.15. 服务宏：

<code>\$SERVICEDESC\$</code>	The long name/description of the service (i.e. "Main Website"). This value is taken from the description directive of the service definition .
<code>\$SERVICEDISPLAYNAME\$</code>	An alternate display name for the service. This value is taken from the display_name directive in the service definition .
<code>\$SERVICESTATE\$</code>	A string indicating the current state of the service ("OK", "WARNING", "UNKNOWN", or "CRITICAL").
<code>\$SERVICESTATEID\$</code>	A number that corresponds to the current state of the service: 0=OK, 1=WARNING, 2=CRITICAL, 3=UNKNOWN.
<code>\$LASTSERVICESTATE\$</code>	A string indicating the last state of the service ("OK", "WARNING", "UNKNOWN", or "CRITICAL").
<code>\$LASTSERVICESTATEID\$</code>	A number that corresponds to the last state of the service: 0=OK, 1=WARNING, 2=CRITICAL, 3=UNKNOWN.
<code>\$SERVICESTATETYPE\$</code>	A string indicating the state type for the current service check ("HARD" or "SOFT"). Soft states occur when service checks return a non-OK state and are in the process of being retried. Hard states result when service checks have been checked a specified maximum number of times.
<code>\$SERVICEATTEMPT\$</code>	The number of the current service check retry. For instance, if this is the second time that the service is being rechecked, this will be the number two. Current attempt number is really only useful when

	writing service event handlers for "soft" states that take a specific action based on the service retry number.
\$MAXSERVICEATTEMPTS\$	The max check attempts as defined for the current service. Useful when writing host event handlers for "soft" states that take a specific action based on the service retry number.
\$SERVICEISVOLATILE\$	Indicates whether the service is marked as being volatile or not: 0 = not volatile, 1 = volatile.
\$SERVICEEVENTID\$	A globally unique number associated with the service's current state. Every time a a service (or host) experiences a state change, a global event ID number is incremented by one (1). If a service has experienced no state changes, this macro will be set to zero (0).
\$LASTSERVICEEVENTID\$	The previous (globally unique) event number that given to the service.
\$SERVICEPROBLEMID\$	A globally unique number associated with the service's current problem state. Every time a service (or host) transitions from an OK or UP state to a problem state, a global problem ID number is incremented by one (1). This macro will be non-zero if the service is currently a non-OK state. State transitions between non-OK states (e.g. WARNING to CRITICAL) do not cause this problem id to increase. If the service is currently in an OK state, this macro will be set to zero (0). Combined with event handlers, this macro could be used to automatically open trouble tickets when services first enter a problem state.
\$LASTSERVICEPROBLEMID\$	The previous (globally unique) problem number that was given to the service. Combined with event handlers, this macro could be used for automatically closing trouble tickets, etc. when a service recovers to an OK state.
\$SERVICELATENCY\$	A (floating point) number indicating the number of seconds that a scheduled service check lagged behind its scheduled check time. For

	instance, if a check was scheduled for 03:14:15 and it didn't get executed until 03:14:17, there would be a check latency of 2.0 seconds.
<code>\$SERVICEEXECUTIONTIME\$</code>	A (floating point) number indicating the number of seconds that the service check took to execute (i.e. the amount of time the check was executing).
<code>\$SERVICEDURATION\$</code>	A string indicating the amount of time that the service has spent in its current state. Format is "XXh YYm ZZs", indicating hours, minutes and seconds.
<code>\$SERVICEDURATIONSEC\$</code>	A number indicating the number of seconds that the service has spent in its current state.
<code>\$SERVICEDOWNTIME\$</code>	A number indicating the current "downtime depth" for the service. If this service is currently in a period of scheduled downtime , the value will be greater than zero. If the service is not currently in a period of downtime, this value will be zero.
<code>\$SERVICEPERCENTCHANGE\$</code>	A (floating point) number indicating the percent state change the service has undergone. Percent state change is used by the flap detection algorithm.
<code>\$SERVICEGROUPNAME\$</code>	The short name of the servicegroup that this service belongs to. This value is taken from the servicegroup_name directive in the servicegroup definition. If the service belongs to more than one servicegroup this macro will contain the name of just one of them.
<code>\$SERVICEGROUPNAMES\$</code>	A comma separated list of the short names of all the servicegroups that this service belongs to.
<code>\$LASTSERVICECHECK\$</code>	This is a timestamp in time_t format (UNIX 系统的秒计数器) indicating the time at which a check of the service was last performed.
<code>\$LASTSERVICESTATECHANGE\$</code>	This is a timestamp in time_t format (UNIX 系统的秒计数器) indicating the time the service last changed state.

<code>\$LASTSERVICEOK\$</code>	This is a timestamp in <code>time_t</code> format (UNIX 系统的秒计数器) indicating the time at which the service was last detected as being in an OK state.
<code>\$LASTSERVICEWARNING\$</code>	This is a timestamp in <code>time_t</code> format (UNIX 系统的秒计数器) indicating the time at which the service was last detected as being in a WARNING state.
<code>\$LASTSERVICEUNKNOWN\$</code>	This is a timestamp in <code>time_t</code> format (UNIX 系统的秒计数器) indicating the time at which the service was last detected as being in an UNKNOWN state.
<code>\$LASTSERVICECRITICAL\$</code>	This is a timestamp in <code>time_t</code> format (UNIX 系统的秒计数器) indicating the time at which the service was last detected as being in a CRITICAL state.
<code>\$SERVICEOUTPUT\$</code>	The first line of text output from the last service check (i.e. "Ping OK").
<code>\$LONGSERVICEOUTPUT\$</code>	The full text output (aside from the first line) from the last service check.
<code>\$SERVICEPERFDATA\$</code>	This macro contains any performance data that may have been returned by the last service check.
<code>\$SERVICECHECKCOMMAND\$</code>	This macro contains the name of the command (along with any arguments passed to it) used to perform the service check.
<code>\$SERVICEACKAUTHOR\$⁸</code>	A string containing the name of the user who acknowledged the service problem. This macro is only valid in notifications where the <code>\$NOTIFICATIONTYPE\$</code> macro is set to "ACKNOWLEDGEMENT".
<code>\$SERVICEACKAUTHORNAME\$⁸</code>	A string containing the short name of the contact (if applicable) who acknowledged the service problem. This macro is only valid in notifications where the <code>\$NOTIFICATIONTYPE\$</code> macro is set to "ACKNOWLEDGEMENT".

<code>\$\$SERVICEACKAUTHORALIAS\$⁸</code>	A string containing the alias of the contact (if applicable) who acknowledged the service problem. This macro is only valid in notifications where the <code>\$\$NOTIFICATIONTYPE\$</code> macro is set to "ACKNOWLEDGEMENT".
<code>\$\$SERVICEACKCOMMENT\$⁸</code>	A string containing the acknowledgement comment that was entered by the user who acknowledged the service problem. This macro is only valid in notifications where the <code>\$\$NOTIFICATIONTYPE\$</code> macro is set to "ACKNOWLEDGEMENT".
<code>\$\$SERVICEACTIONURL\$</code>	Action URL for the service. This macro may contain other macros (e.g. <code>\$\$HOSTNAME\$</code> or <code>\$\$SERVICEDESC\$</code>), which can be useful when you want to pass the service name to a web page.
<code>\$\$SERVICENOTESURL\$</code>	Notes URL for the service. This macro may contain other macros (e.g. <code>\$\$HOSTNAME\$</code> or <code>\$\$SERVICEDESC\$</code>), which can be useful when you want to pass the service name to a web page.
<code>\$\$SERVICENOTES\$</code>	Notes for the service. This macro may contain other macros (e.g. <code>\$\$HOSTNAME\$</code> or <code>\$\$SERVICESTATE\$</code>), which can be useful when you want to service-specific status information, etc. in the description

表 8.16. 服务组宏：⁶

<code>\$\$SERVICEGROUPALIAS\$⁶</code>	The long name / alias of either 1) the servicegroup name passed as an on-demand macro argument or 2) the primary servicegroup associated with the current service (if not used in the context of an on-demand macro). This value is taken from the alias directive in the servicegroup definition .
<code>\$\$SERVICEGROUPMEMBERS\$⁶</code>	A comma-separated list of all services that belong to either 1) the servicegroup name passed as an on-demand macro argument or 2) the primary servicegroup associated with the current service (if not used in the context of an on-demand macro).

<code>\$SERVICEGROUPNOTES\$⁶</code>	The notes associated with either 1) the servicegroup name passed as an on-demand macro argument or 2) the primary servicegroup associated with the current service (if not used in the context of an on-demand macro). This value is taken from the notes directive in the servicegroup definition .
<code>\$SERVICEGROUPNOTESURL\$⁶</code>	The notes URL associated with either 1) the servicegroup name passed as an on-demand macro argument or 2) the primary servicegroup associated with the current service (if not used in the context of an on-demand macro). This value is taken from the notes_url directive in the servicegroup definition .
<code>\$SERVICEGROUPNOTES\$⁶</code>	The action URL associated with either 1) the servicegroup name passed as an on-demand macro argument or 2) the primary servicegroup associated with the current service (if not used in the context of an on-demand macro). This value is taken from the action_url directive in the servicegroup definition .

表 8.17. 联系人宏:

<code>\$CONTACTNAME\$</code>	Short name for the contact (i.e. "jdoe") that is being notified of a host or service problem. This value is taken from the contact_name directive in the contact definition .
<code>\$CONTACTALIAS\$</code>	Long name/description for the contact (i.e. "John Doe") being notified. This value is taken from the alias directive in the contact definition .
<code>\$CONTACTEMAIL\$</code>	Email address of the contact being notified. This value is taken from the email directive in the contact definition .
<code>\$CONTACTPAGER\$</code>	Pager number/address of the contact being notified. This value is taken from the pager directive in the contact definition .
<code>\$CONTACTADDRESSn\$</code>	Address of the contact being notified. Each contact can have six different

	addresses (in addition to email address and pager number). The macros for these addresses are \$CONTACTADDRESS1\$ – \$CONTACTADDRESS6\$. This value is taken from the addressx directive in the contact definition .
\$CONTACTGROUPNAME\$	The short name of the contactgroup that this contact is a member of. This value is taken from the contactgroup_name directive in the contactgroup definition . If the contact belongs to more than one contactgroup this macro will contain the name of just one of them.
\$CONTACTGROUPNAMES\$	A comma separated list of the short names of all the contactgroups that this contact is a member of.

表 8.18. 联系人组宏：⁵

\$CONTACTGROUPALIAS\$ ⁷	The long name / alias of either 1) the contactgroup name passed as an on-demand macro argument or 2) the primary contactgroup associated with the current contact (if not used in the context of an on-demand macro). This value is taken from the alias directive in the contactgroup definition .
\$CONTACTGROUPMEMBERS\$ ⁷	A comma-separated list of all contacts that belong to either 1) the contactgroup name passed as an on-demand macro argument or 2) the primary contactgroup associated with the current contact (if not used in the context of an on-demand macro).

表 8.19. 汇总统计宏：

\$TOTALHOSTSUP\$	This macro reflects the total number of hosts that are currently in an UP state.
\$TOTALHOSTSDOWN\$	This macro reflects the total number of hosts that are currently in a DOWN state.
\$TOTALHOSTSUNREACHABLE\$	This macro reflects the total number of hosts that are

	currently in an UNREACHABLE state.
\$TOTALHOSTSDOWNUNHANDLED\$	This macro reflects the total number of hosts that are currently in a DOWN state that are not currently being "handled". Unhandled host problems are those that are not acknowledged, are not currently in scheduled downtime, and for which checks are currently enabled.
\$TOTALHOSTSUNREACHABLEUNHANDLED\$	This macro reflects the total number of hosts that are currently in an UNREACHABLE state that are not currently being "handled". Unhandled host problems are those that are not acknowledged, are not currently in scheduled downtime, and for which checks are currently enabled.
\$TOTALHOSTPROBLEMS\$	This macro reflects the total number of hosts that are currently either in a DOWN or an UNREACHABLE state.
\$TOTALHOSTPROBLEMSUNHANDLED\$	This macro reflects the total number of hosts that are currently either in a DOWN or an UNREACHABLE state that are not currently being "handled". Unhandled host problems are those that are not acknowledged, are not currently in scheduled downtime, and for which checks are currently enabled.
\$TOTALSERVICESOK\$	This macro reflects the total number of services that are currently in an OK state.
\$TOTALSERVICESWARNING\$	This macro reflects the total number of services that are currently in a WARNING state.
\$TOTALSERVICESCRITICAL\$	This macro reflects the total number of services that are currently in a CRITICAL state.
\$TOTALSERVICESUNKNOWN\$	This macro reflects the total number of services that are currently in an UNKNOWN state.

<code>\$TOTALSERVICESWARNINGUNHANDLED\$</code>	This macro reflects the total number of services that are currently in a WARNING state that are not currently being "handled". Unhandled services problems are those that are not acknowledged, are not currently in scheduled downtime, and for which checks are currently enabled.
<code>\$TOTALSERVICESCRITICALUNHANDLED\$</code>	This macro reflects the total number of services that are currently in a CRITICAL state that are not currently being "handled". Unhandled services problems are those that are not acknowledged, are not currently in scheduled downtime, and for which checks are currently enabled.
<code>\$TOTALSERVICESUNKNOWNUNHANDLED\$</code>	This macro reflects the total number of services that are currently in an UNKNOWN state that are not currently being "handled". Unhandled services problems are those that are not acknowledged, are not currently in scheduled downtime, and for which checks are currently enabled.
<code>\$TOTALSERVICEPROBLEMS\$</code>	This macro reflects the total number of services that are currently either in a WARNING, CRITICAL, or UNKNOWN state.
<code>\$TOTALSERVICEPROBLEMSUNHANDLED\$</code>	This macro reflects the total number of services that are currently either in a WARNING, CRITICAL, or UNKNOWN state that are not currently being "handled". Unhandled services problems are those that are not acknowledged, are not currently in scheduled downtime, and for which checks are currently enabled.

表 8.20. 通知宏:

<code>\$NOTIFICATIONTYPE\$</code>	A string identifying the type of notification that is being sent ("PROBLEM", "RECOVERY", "ACKNOWLEDGEMENT", "FLAPPINGSTART", "FLAPPINGSTOP", "FLAPPINGDISABLED", "DOWNTIMESTART",
-----------------------------------	---

	"DOWNTIMEEND", or "DOWNTIMECANCELLED").
\$NOTIFICATIONRECIPIENTS\$	A comma-separated list of the short names of all contacts that are being notified about the host or service.
\$NOTIFICATIONISESCALATED\$	An integer indicating whether this was sent to normal contacts for the host or service or if it was escalated. 0 = Normal (non-escalated) notification , 1 = Escalated notification.
\$NOTIFICATIONAUTHOR\$	A string containing the name of the user who authored the notification. If the \$NOTIFICATIONTYPE\$ macro is set to "DOWNTIMESTART" or "DOWNTIMEEND", this will be the name of the user who scheduled downtime for the host or service. If the \$NOTIFICATIONTYPE\$ macro is "ACKNOWLEDGEMENT", this will be the name of the user who acknowledged the host or service problem. If the \$NOTIFICATIONTYPE\$ macro is "CUSTOM", this will be name of the user who initiated the custom host or service notification.
\$NOTIFICATIONAUTHORNAME\$	A string containing the short name of the contact (if applicable) specified in the \$NOTIFICATIONAUTHOR\$ macro.
\$NOTIFICATIONAUTHORALIAS\$	A string containing the alias of the contact (if applicable) specified in the \$NOTIFICATIONAUTHOR\$ macro.
\$NOTIFICATIONCOMMENT\$	A string containing the comment that was entered by the notification author. If the \$NOTIFICATIONTYPE\$ macro is set to "DOWNTIMESTART" or "DOWNTIMEEND", this will be the comment entered by the user who scheduled downtime for the host or service. If the \$NOTIFICATIONTYPE\$ macro is "ACKNOWLEDGEMENT", this will be the comment entered by the user who acknowledged the host or service problem. If the \$NOTIFICATIONTYPE\$ macro is "CUSTOM", this will be comment entered by the user who initiated the custom host or service notification.

\$HOSTNOTIFICATIONNUMBER\$	<p>The current notification number for the host. The notification number increases by one (1) each time a new notification is sent out for the host (except for acknowledgements). The notification number is reset to 0 when the host recovers (after the recovery notification has gone out). Acknowledgements do not cause the notification number to increase, nor do notifications dealing with flap detection or scheduled downtime.</p>
\$HOSTNOTIFICATIONID\$	<p>A unique number identifying a host notification. Notification ID numbers are unique across both hosts and service notifications, so you could potentially use this unique number as a primary key in a notification database. Notification ID numbers should remain unique across restarts of the Nagios process, so long as you have state retention enabled. The notification ID number is incremented by one (1) each time a new host notification is sent out, and regardless of how many contacts are notified.</p>
\$SERVICENOTIFICATIONNUMBER\$	<p>The current notification number for the service. The notification number increases by one (1) each time a new notification is sent out for the service (except for acknowledgements). The notification number is reset to 0 when the service recovers (after the recovery notification has gone out). Acknowledgements do not cause the notification number to increase, nor do notifications dealing with flap detection or scheduled downtime.</p>
\$SERVICENOTIFICATIONID\$	<p>A unique number identifying a service notification. Notification ID numbers are unique across both hosts and service notifications, so you could potentially use this unique number as a primary key in a notification database. Notification ID numbers should remain unique across restarts of the Nagios process, so long as you have state retention enabled. The notification ID number is incremented by one (1) each time a new service notification is sent out, and</p>

	regardless of how many contacts are notified.
--	---

表 8.21. 日期/时间宏:

<code>\$LONGDATETIME\$</code>	当前的日期/时间戳(如 Fri Oct 13 00:30:28 CDT 2000)。日期的格式符合 date format 域设置。
<code>\$SHORTDATETIME\$</code>	当前的日期/时间戳(如 10-13-2000 00:30:28)。日期的格式符合 date format 域设置。
<code>\$DATE\$</code>	日期戳(如 10-13-2000)。日期格式符合 date format 域设置。
<code>\$TIME\$</code>	当前时间戳(如 00:30:28)。
<code>\$TIMET\$</code>	以 <code>time_t</code> 格式表示的当前时间戳(UNIX 系统的秒计数器)。
<code>\$ISVALIDTIME:\$⁹</code>	<p>这是按需宏所特有的, 返回 1 或 0, 用以指示一个指定时刻在指定时间周期对象定义下是否合法。有两种方式来使用这个宏:</p> <ul style="list-style-type: none"> • \$ISVALIDTIME:24x7\$——如果当前时间点在“24x7”这个时间周期定义里是个合法时间则返回“1”, 如果不是合法时间则返回“0”; • \$ISVALIDTIME:24x7:timestamp\$——如果由“timestamp”(必须是 <code>time_t</code> 格式)所指定的时刻在“24x7”这个时间周期定义里是个合法时间则返回“1”, 否则返回“0”。
<code>\$NEXTVALIDTIME:\$⁹</code>	<p>这是按需宏所特有的, 返回在指定的时间周期对象定义下的下一个合法的时间值(是 <code>time_t</code> 格式的), 有两种方式来使用该宏:</p> <ul style="list-style-type: none"> • \$NEXTVALIDTIME:24x7\$——将返回在“24x7”这个时间周期对象定义所限定时间段内的自当前时间后的下一个合法时间点。 • \$NEXTVALIDTIME:24x7:timestamp\$——将返回在“24x7”这个时间周期对象定义所限定时间段内的从“timestamp”(是个 <code>time_t</code> 格式的)所标时间后的下一个合法时间点。 <p>如果在指定的时间周期对象定义里没能找到下一个合法时间, 宏将被置为“0”。</p>

表 8.22. 文件宏:

\$MAINCONFIGFILE\$	主配置文件 的保存位置。
\$STATUSDATAFILE\$	状态数据文件 的保存位置。
\$COMMENTDATAFILE\$	注释数据文件的保存位置。
\$DOWNTIMedatafile\$	停机时间数据文件的保存位置。
\$RETENTIONDATAFILE\$	状态保持数据文件 的保存位置。
\$OBJECTCACHEFILE\$	对象缓存文件 的保存位置。
\$TEMPFILE\$	The location of the 临时文件 的保存位置。
\$TEMPPATH\$	临时目录 变量所指向的目录。
\$LOGFILE\$	日志文件 的保存位置。
\$RESOURCEFILE\$	资源文件 的保存位置。
\$COMMANDFILE\$	命令文件 的保存位置。
\$HOSTPERFDATAFILE\$	(如果定义过的) 主机性能数据文件的保存位置。
\$SERVICEPERFDATAFILE\$	(如果定义过的) 服务性能数据文件的保存位置。

表 8.23. 其他宏：

\$PROCESSSTARTTIME\$	以 time_t (UNIX 系统的秒计数器) 格式的时间戳指向了 Nagios 进程启动或重启的时刻。可以用\$TIMET\$减去\$PROCESSSTARTTIME\$来计算出 Nagios 自最后一次启动至今共运行了多少秒。
\$EVENTSTARTTIME\$	以 time_t 格式的时间戳，指示了 Nagios 开始处理事件(检测等)的时刻。可以用\$EVENTSTARTTIME\$减去\$PROCESSSTARTTIME\$来计算出 Nagios 花了多少秒来启动就绪。
\$ADMINEMAIL\$	全局的管理员Email地址，这个值是从配置文件里的 admin_email 域里取得的。
\$ADMINPAGER\$	全局管理员的BP机号或地址，这个是从 admin_pager 域里取得的值。
\$ARGn\$	指向第 n 个命令传递参数(通知、事件处理、服务检测等)。Nagios 支持最多 32 个

	参数宏 (从\$ARG1\$到\$ARG32\$)。
\$USERn\$	指向第n个用户的宏。用户宏可以在 资源文件 里定义一个或多个。Nagios支持最多32 个用户宏 (从\$USER1\$到\$USER32\$)。

8.3.4. 注意

¹ 当主机处于检测状态时与之相关的宏是无效的 (如他们没有被检测也就还没有定性状态时)；

² 当服务处于检测状态时与之相关的宏是无效的 (如他们没有被检测也就还没有定性状态时)；

³ 当主机宏被用于服务相关命令时 (如服务通知、事件处理等) 主机宏被指向了与服务相关的主机；

⁴ 当主机与服务汇总统计宏被用于通知命令时，只是当联系人被授权的主机或服务被统计到汇总结果之中 (如主机和服务配置以该联系人为通知接收人的情况)；

⁵ 这些宏通常是指向当前主机所属的第一个 (首要) 主机组。很多情况下可被认为是一种主机宏。然而这些宏不能做为按需宏里的主机宏，当你用这些宏传主机组名时这些宏可被用做按需宏的主机组宏。如：
\$HOSTGROUPMEMBERS:hg1\$将返回主机组 **hg1** 里的全部成员主机，是个以逗号分开的列表。

⁶ 这些宏通常是指向当前服务所属的第一个 (首要) 服务组。很多情况下可被认为是一种服务宏。然而这些宏不能做为按需宏里的服务宏，当你用这些宏传服务组名时这些宏可被用做按需宏的服务组宏。如：
\$SERVICEGROUPMEMBERS:sg1\$将返回服务组 **sg1** 里的全部成员服务，是个以逗号分开的列表。

⁷ 这些宏通常是指向当前联系人所属的第一个 (首要) 联系人组。很多情况下可被认为是一种联系人宏。然而这些宏不能做为按需宏里的联系人宏，当你用这些宏传联系人组名时这些宏可被用做按需宏的联系人宏。如：\$CONTACTGROUPMEMBERS:cg1\$将返回联系人组 **cg1** 里的全部成员联系人，是个以逗号分开的列表。

⁸ 尽量不使用这些宏。用更通用的宏\$NOTIFICATIONAUTHOR\$、\$NOTIFICATIONAUTHORNAME\$、\$NOTIFICATIONAUTHORALIAS\$或\$NOTIFICATIONAUTHORCOMMENT\$等宏替换。

⁹ 这些宏只用于按需宏 — 也就是说为了使用它们必须要提供额外的参数。这些宏在环境变量中不可用。

¹⁰ 汇总统计宏在当设置 [use_large_installation_tweaks](#)选项使能时在环境变量中不可用，因为这将非常密集使用CPU来计算；

8.4. 如何确认网络中主机的状态与可达性

8.4.1. 介绍

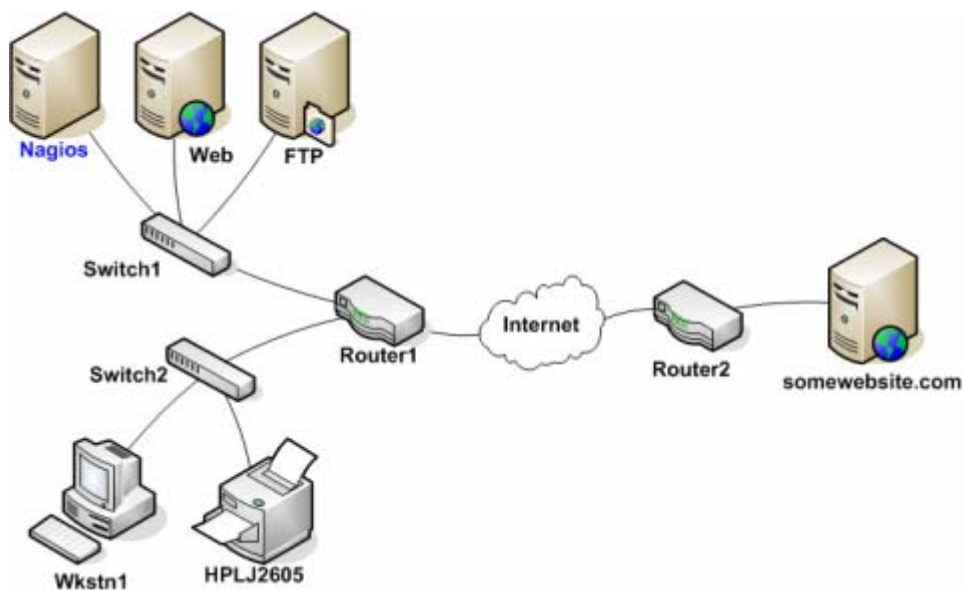
如果做过技术支持就会有这种困惑，用户抱怨说“因特网不通了”而你却很抓狂。做为一个负责任的人，可以肯定的是没有人会拉掉网络供电电源，但是，由于用户在办公室上不了网却确实地存在。

如果是个技术性故障，可能会找寻故障问题所在。可能会重启动用户计算机，可能是用户的网线头没插好，也可能是核心路由器有点“抽风”。无论哪个问题，只有一个肯定是肯定存在的——因特网不通。只是对那个用户而言因特网是不可达的。

Nagios 具备判断所监控主机是否处于宕机还是不可达状态的能力。两个是很不同的状态(虽然它们是相关联的)并且可以帮助你快速找到故障根源。下面是网络可达性逻辑如何来分辨两种状态的说明...

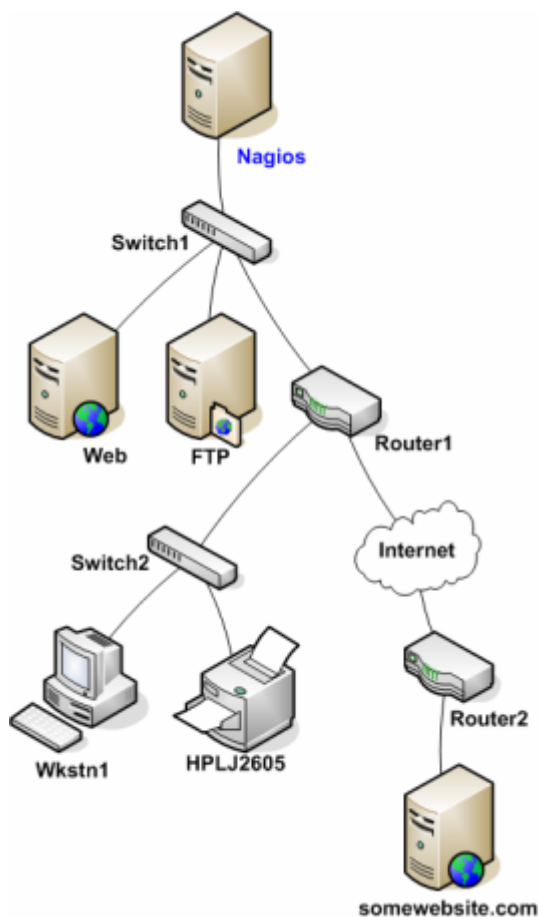
8.4.2. 样板网络

下面是一个简易的网络图。在这个例子中，假定监控了图中全部的主机(服务器、路由器和交换机等)。Nagios 安装并运行在图中名为 **Nagios** 主机上。



8.4.3. 定义网络主机的父子关系

为使 Nagios 分辨出所监控主机所处于宕机还是不可达状态，必须要给出主机间的联接关系——联接关系要基于 Nagios 主守护程序所在点为根点。追踪每个从 Nagios 主守护程序到各自节点的数据包将可以得到这种关系。每个交换机、路由器和服务器上的数据包碰撞或通过都认为是网络拓扑中的一跳“hop”，需要在 Nagios 里定义出主机间的父/子节点关系，下面给出例子中的网络在 Nagios 中的父/子关系视图：



看图可以知道各个被监控主机的父/子节点关系了，但在Nagios的配置里如何来表达呢？可以用 [主机对象定义](#) 里面的 **parents** 域来实现。下面是例子中的对象定义的关于父/子节点关系的片段：

```

define host{
    host_name            Nagios    ; <-- The local host has no parent - it is the
topmost host
}

define host{
    host_name            Switch1
    parents              Nagios
}

define host{
    host_name            Web
    parents              Switch1
}

define host{
    host_name            FTP

```

```

        parents          Switch1
    }

define host{
    host_name            Router1
    parents              Switch1
}

define host{
    host_name            Switch2
    parents              Router1
}

define host{
    host_name            Wkstn1
    parents              Switch2
}

define host{
    host_name            HPLJ2605
    parents              Switch2
}

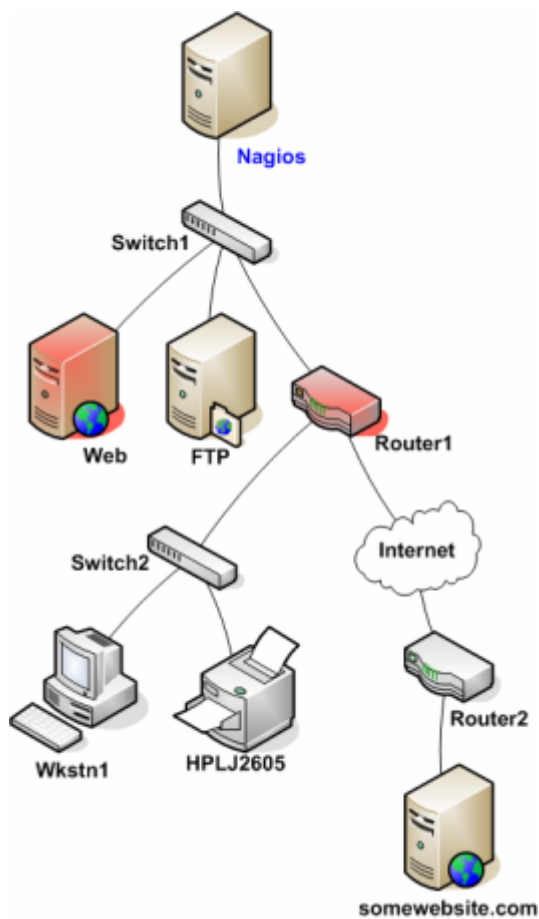
define host{
    host_name            Router2
    parents              Router1
}

define host{
    host_name            somewebsite.com
    parents              Router2
}

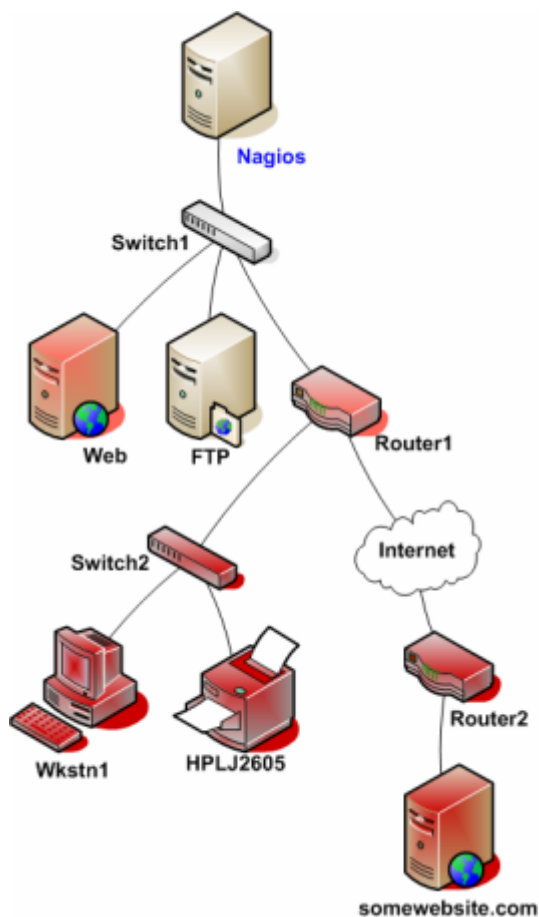
```

8.4.4. 可达性逻辑的运转

现在已经将主机的父/子逻辑关系正确地配置到了 Nagios 里，下面看一下当故障产生时会发生什么事。假定两个主机—**Web** 与 **Router1**—掉线了...



当主机状态改变(如从运行到宕机), Nagios 唤起了网络可达性逻辑。可达性逻辑将初始化一个并发检测, 只要是状态改变的主机的父/子节点都会被检测。在网络框架里变化发生时, 这将使得 Nagios 迅速地对当前网络状态进行分析判定。



在本例中，Nagios 将判定 **Web** 和 **Router1** 都处于宕机状态因为到达这两台主机的“路径”并没有阻塞。

Nagios 将判定出在拓扑逻辑上 **Router1** 之下的所有主机处于不可达状态，因为 Nagios 无法找到它们。**Router1** 的宕机将阻塞了到达这些主机的路径。这些主机可能运行得好着呢，也或是已经掉线—Nagios 无法得知因为无法把测试包送达那里，因而 Nagios 认为那些主机是不可达而不是宕机。

8.4.5. 不可达状态与通知

默认情况下 Nagios 将会对主机处于宕机和不可达状态时都会送出通知给对应的联系人。如果是管理员或技术支持人员，人可能不想接到不可达状态主机的通知。你了解所处网络的拓扑结构，当 Nagios 通知路由器或防火墙宕机时，肯定的是在之后的主机都会不可达。

如果你想避开由于网络状态改变而导致的主机不可达的事件风暴，可以在 [主机对象定义](#)里的 `notification_options`域中排除“不可达”状态(u)，同时，或者是也可以将 [联系人对象定义](#)里的 `host_notification_options`域里排除“不可达”状态(u)。

8.5. 可变服务

8.5.1. 介绍

Nagios 具备分辨“通常服务”与“可变服务”的能力。在服务对象定义里的 `is_volatile` 域可以指定该服务是否是一个可变服务。很多情况下，绝大多数的被监控服务都不是“可变服务”(也就是“通常服务”)，而可变服务可在几种情形下使用...

8.5.2. 可用于什么情况？

可变服务可用于监控...

- 每次检测时状态时都会被自动地复位到“正常”状态；
- 每次必须要给出关注的某个故障事件象安全性事件(不仅仅是第一次才需要注意)

8.5.3. 可变服务有何特别之处？

在每次处于 [硬态](#) 非正常状态情况下做检测而检测将返回一个非正常状态(如没有发生状态变换)的情况下，可变服务与通常服务将会有三个明显差异：

- 非正常服务状态被记录下来(产生日志)；
- 服务故障将会送给联系人(如果 [该故障需要做通知的话](#))。注意：在可变服务对象的设置里的通知间隔将被忽略；
- 针对该服务的 [事件处理](#) 将会被执行(如果对象里定义过的话)。

当指定的服务处于非正常状态并且一个硬态状态变换刚刚发生的时候，以上操作才会发生。也就是说，只是该服务首次转入相同的非正常状态时才会做。如果之后的服务检测结果同样是非正常状态，没有硬态状态变换发生，那么就不会再有以上操作。

提示



如果只是希望产生日志，请考虑换用 [状态追踪](#) 的功能选项。

8.5.4. The Power Of Two

如果将可变服务与 [强制服务检测](#) 两个功能特性组合使用，可以实现很有用的功能。下面的例子中包括处理SNMP Trap告警、安全警报等。

给个例子... 假定你运行了 [PortSentry\(端口哨兵\)](#) 来检测对你机器的端口扫描和防火墙的入侵企图。如果要Nagios来显示端口扫描，可以照下面的来做...

Nagios 配置：

- 创建一个服务对象定义，命名为 **Port Scans** 并且与那个运行 PortSentry 的主机绑定在一起；
- 设定服务对象里的 `max_check_attempts` 域值为 1。这将通知Nagios当一个非正常状态报告时服务状态将即刻转入 [硬态](#) 状态(不必再重试)；

- 设定服务对象里的 **active_checks_enabled** 域值为 0。这会阻止 Nagios 不必再启动针对服务的自主检测；
- 设定服务对象里的 **passive_checks_enabled** 域值为 1。这将会启动针对该服务的强制检测；
- 设置服务对象里的 **is_volatile** 域值为 1。

PortSentry 配置：

编辑 PortSentry 配置文件 (portsentry.conf) 并且定义一个 **KILL_RUN_CMD** 命令，象这样：

```
KILL_RUN_CMD="/usr/local/Nagios/libexec/eventhandlers/submit_check_result host_name
'Port Scans' 2 'Port scan from host $TARGET$ on port $PORT$. Host has been firewalled.'"
```

要确保把里面的 **host_name** 替换为服务绑定的主机对象的短名称。

端口扫描脚本：

创建一个 SHELL 脚本，放在 **/usr/local/nagios/libexec/eventhandlers** 目录下并命名为 **submit_check_result**。这个脚本的内容可能会是这样的...

```
#!/bin/sh

# Write a command to the Nagios command file to cause
# it to process a service check result

echocmd="/bin/echo"

CommandFile="/usr/local/nagios/var/rw/nagios.cmd"

# get the current date/time in seconds since UNIX epoch
datetime=`date +%s`

# create the command line to add to the command file
cmdline="[$datetime] PROCESS_SERVICE_CHECK_RESULT;$1;$2;$3;$4"

# append the command to the end of the command file
`$echocmd $cmdline >> $CommandFile`
```

那么当 PortSentry 检测到了一个针对机器的端口扫描时将会做些什么呢？

- PortSentry 将会防护该主机(这是 PortSentry 软件的功能)；
- PortSentry 将执行 **submit_check_result** SHELL 脚本并送给 Nagios 一个强制检测的结果；
- Nagios 将从外部命令文件中读取并确认由 PortSentry 提交的内容；
- Nagios 将把 **Port Scans** 服务状态置成“硬态紧急状态”并给联系人送出通知。

很棒吧？

8.6. 主机与服务的刷新检测



8.6.1. 介绍

Nagios 有对主机和服务检测的结果做“刷新检测”的特性。刷新检测的目的是为保证由外部应用而做的主机与服务强制检测可以正常提供结果数据。

刷新检测在确保频繁地接收 [强制检测](#) 结果时很有用。它在 [分布式](#) 和 [冗余式失效性](#) 监控环境下非常有用。

8.6.2. 刷新检测如何工作？

Nagios 定期地刷新全部的打开检测功能的主机与服务检测状态。

- 由每个主机或服务计算出一个刷新门限；
- 对于每个主机与服务，最后一次检测结果的时间长短会与刷新门限相比对；
- 如果最后一次检测结果的时间大于刷新检测门限，检测结果会被认为是“陈旧”的；
- 如果检测结果被认为是“陈旧的”，Nagios 将强制地针对该主机或服务用主机与服务对象定义里指定的命令来执行一次 [自主检测](#)。

提示



一次自主检测总是被执行，即便是自主检测在程序层面或是主机的与服务的指定自主检测选项被关闭；

例如，如果一个服务的刷新门限设定为 60 秒，Nagios 将认为如果最后一次检测结果如果存在时间超过 60 秒将会认为该结果是“陈旧”的。

8.6.3. 使能刷新检测

如果要打开刷新检测需要做如下事情：

- 在程序层面使能刷新检测要用 [check_service_freshness](#) 和 [check_host_freshness](#) 域来控制；
- 用 [service_freshness_check_interval](#) 和 [host_freshness_check_interval](#) 选项来设置 Nagios 以何频度来刷新主机和服务检测结果；
- 在主机与服务对象定义里打开主机的和服务的刷新检测开关，是设置对象里的 `check_freshness` 选项值为 1；
- 配置主机和服务对象定义里的刷新检测门限，即设置对象里的 `freshness_threshold` 选项；

- 配置主机与服务对象定义里的 **check_command** 选项指向一个合法的可被用于自主检测的命令，当发现结果“陈旧”时可以使用该命令；
- 在主机与服务对象定义里的 **check_period** 选项可用于当 Nagios 认为需要进行一次刷新时可用时间周期，因而要保证它是一个合法的时间周期(译者注—在需要自主检测时刻可落入该时间周期定义)；

提示



如果没有指定一个主机的或服务的刷新门限 **freshness_threshold** 值(或是把它设置为 0)，Nagios 将自动地计算门限，它是基于以何频度来监控特定的主机或服务。推荐是清楚地定义出刷新门限，而不是让 Nagios 来自主决定它。

8.6.4. 样例

下面是一个可能需要刷新检测的服务样例，它是每天夜间做备份作业的服务。可能已经有一个外部脚在作业完成时向 Nagios 提交备份作业的结果。在这种情形下，全部的针对该服务的检测与结果将是由强制检测的外部应用来完成的。为保证每天的备份作业的状态都会被 Nagios 所收集报告，需要打开针对该服务的刷新检测。如果外部对备份作业脚本没有提交检测结果，可以让 Nagios 取得一个紧急处置结果，象这样...

下面是该服务定义的样本(有些东西被省略了)...

```
define service{
    host_name                backup-server
    service_description      ArcServe Backup Job
    active_checks_enabled    0                ; active checks are NOT enabled
    passive_checks_enabled   1                ; passive checks are enabled (this is how
results are reported)
    check_freshness          1
    freshness_threshold       93600           ; 26 hour threshold, since backups may not
always finish at the same time
    check_command             no-backup-report ; this command is run only if the service
results are "stale"
    ...other options...
}
```

应该注意，该服务的自主检测是关闭的，这是因为该服务的检测是由外部应用使用强制检测机制送达 Nagios。刷新检测打开了而且刷新门限设置为 26 小时。这个设置略长于备份作业每天所需要的 24 小时，因为备份作业每天时间长短不同（它是由多少数据量要做备份和当时的网络拥塞等等情况所决定）。设定的 **no-backup-report** 命令只是当服务检测结果被认为是“陈旧”的时候才执行的，这个 **no-backup-report** 命令的定义看起来象是这样：

```
define command{  
    command_name      no-backup-report  
    command_line      /usr/local/nagios/libexec/nobackupreport.sh  
}
```

这个 **nobackupreport.sh** 脚本放在 **/usr/local/nagios/libexec** 目录里，内容可能是这样的：

```
#!/bin/sh  
  
/bin/echo "CRITICAL: Results of backup job were not reported!"  
  
exit 2
```

如果 Nagios 检测到服务结果是“陈旧”的，它会以自主检测的方式来运行 **no-backup-report** 命令，也就是执行 **/usr/local/nagios/libexec/nobackupreport.sh** 脚本，它将给 Nagios 返回一个紧急状态。那么这个备份作业的服务就将处于紧急状态（如果它还不是紧急状态的话）同时相关人员可能会收到一个故障通知。

8.7. 感知和处理状态抖动

8.7.1. 介绍

Nagios 支持可选的发现主机与服务抖动的功能。当服务与主机状态改变过于频繁时会产生抖动，其结果产生了故障与恢复的通知风暴。抖动可能是由于配置的问题（如门限过低）、有毛病的服务或是真实的网络问题。

8.7.2. 感知抖动是如何工作的？

在此之前，我想说的是抖动的感知有点难实现。如何精确地确定网络与主机的什么叫做“过分频繁”？当我第一次考虑对感知抖动的实现时，我试图找到发现抖动本该或应该或是如何做的信息，但是一无所获，所以决定用一种对我言是一种合理的方式来解决它...

每当 Nagios 对主机与服务进行检测，它将查看该主机或服务是否已开始或停止抖动，条件有几条：

- 保存好的对主机与服务的检测结果至少 21 个；
- 分析历史检测结果确定状态变换发生了；

- 用状态转换判定主机与服务状态值改变的百分比；
- 比较这个百分比是否越过了设定的抖动门限的最低值与最高值；

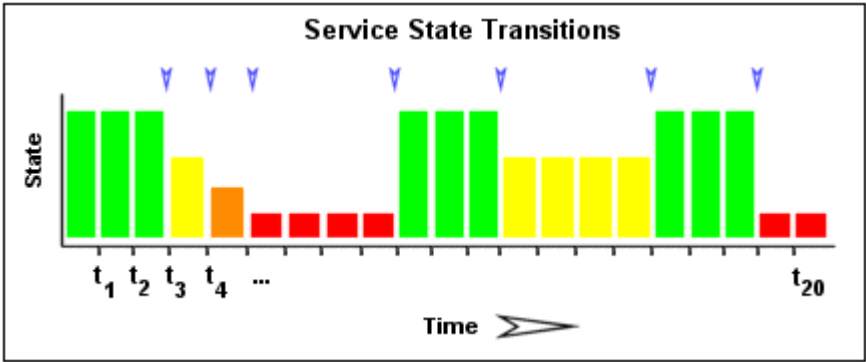
认定主机与服务的**抖动开始**是它的状态改变率首次**高于抖动门限的高限**。

认定主机与服务的**抖动结束**是它的状态改变率**低于抖动门限低限**(前提是它已经处于抖动状态)。

8.7.3. 例子

下面用个服务来更详细地说明如何感知抖动的...

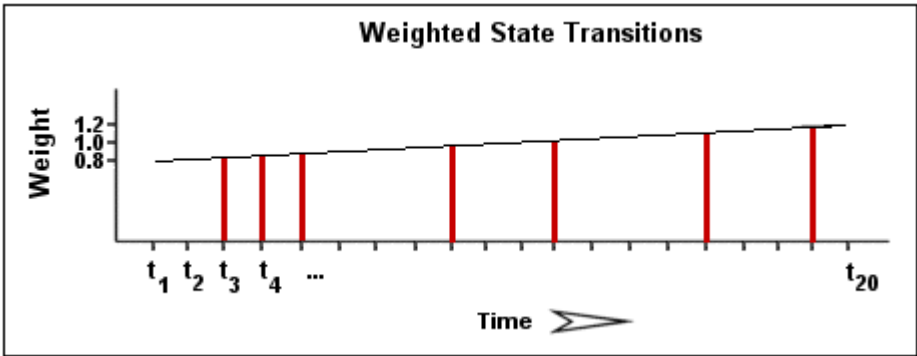
下图给出了最近 21 次检测结果的按时序的历史状态。正常 (OK) 态标记为绿色，告警 (WARNING) 态为黄色，紧急 (CRITICAL) 为红色，未知 (UNKNOWN) 态为橙色。



对历史检测结果的检查决定了哪个时间里有状态变换发生，状态变换发生于存档状态与其前一次状态不同的时刻。由于用数组保存了最近 21 次检测结果，因而可以知道最多可能会产生 20 次变化。在本例中有 7 次状态变化，在图中上方用蓝色箭头示意出来。

感知状态抖动逻辑使用状态变换来判定整体服务的状态变化率，用于度量服务变化或更改的频度。没有发生过状态变化的变化率为 0%，而每次都变化的状态变化率是 100%。服务的状态变化应该在此之间变化。

当计算服务的状态变化率时，感知抖动的算法将会给对近期变化更多权重，旧的变化权重低。特别地，将近期变化给出 50% 的权重。图中示出对指定服务使用了近期变化有更多权重来计算整体变化率的情况。



利用图示结果，计算一下服务的状态变化率。共有 7 次状态变化(分别位于 $t_3, t_4, t_9, t_{12}, t_{16}$ 和 t_{19})。没有任何状态变化权重时结果将会是 35%:

$$(7 \text{ 次查出的状态变化} / 20 \text{ 次最大状态变化次数}) * 100\% = 35\%$$

因为感知抖动的检测逻辑使用近期变化更大的权重，所以该例中实际计算时变化率会低于 35%。假定这个加权后的变化率是 31%...

使用计算后的服务的状态变化率 (31%) 来比对抖动门限将会发生：

- 如果先前**没有**发生抖动且 31%**等于或超出了**抖动门限的高限，Nagios 将判定服务开始抖动；
- 如果服务先前**处于**抖动而且 31%**低于**抖动门限的低限，Nagios 将判定服务停止抖动；

如果两个都没有发生，感知抖动逻辑将不会对服务做任何动作，因为它既没有变为抖动也或许正在抖动。

8.7.4. 服务的抖动感知

每当 Nagios 对服务进行检测时就会来做检查看它是否抖动 (不管是自主检测还是强制检测)。

服务的抖动感知机制见上面例子中的描述说明。

8.7.5. 主机的抖动感知

主机的抖动感知与服务的相似，只是一个重要的不同：Nagios 将在如下情形时尝试对其进行抖动中的检测：

- 主机检测时 (自主检测或强制检测时都会做)
- 有时与主机绑定的服务被检测时。更特殊地，当至少 **x** 次的抖动感知做过时，此处的 **x** 等于全部与主机绑定服务的平均检测间隔时间。

为何要这样？由于最少的两次抖动检查次数间的时间最少是等于服务检测间隔时间。然而可能对主机的监控并非基于规格化的间隔，所以对主机的抖动检测可能对它的抖动感知的检查不是主机检测的间隔时间。同样地，要知道对服务的检查会叠加到主机的抖动感知检测上。毕竟服务是主机上的属性而不是别的... 在种种检查速率相比之下，这个是最好的方式来多次地对主机进行抖动检查，所以你也得如此。

8.7.6. 抖动检测门限

Nagios 在抖动感知逻辑中用若干个值来判定状态变化率。既有主机的也有服务的，配置里面有**全局**的门限高限和低限也有专门针对**主机的**或是**服务的**的门限。Nagios 将在没有指定专门主机的或服务的门限时使用全局的门限值。

下表给出了全局的、专给主机的和专给服务的的门限值的控制变量。

表 8.24.

对象类型	全局变量	对象专属的变量
主机	low host flap threshold high host flap threshold	low flap threshold high flap threshold

对象类型	全局变量	对象专属的变量
服务	low_service_flap_threshold high_service_flap_threshold	low_flap_threshold high_flap_threshold

8.7.7. 给抖动检测所用的状态

通常 Nagios 将记录下针对主机和服务的最后 21 次检测结果用于抖动感知逻辑，而不管全部的检查结果。

提示



在抖动感知逻辑中可以排除主机或服务的某种状态，在主机或服务定义中使用 **flap_detection_options** 域来指明哪些状态(如运行(UP)、宕机(DOWN)、正常(OK)、紧急(CRITICAL)等)要进入抖动检查。如果没有设置它，全部的主机与服务状态都会被用于抖动感知逻辑之中。

8.7.8. 抖动处理

当服务或主机首次发现处于抖动时，Nagios 将会：

- 记录下服务与主机正在抖动的信息；
- 给主机与服务增加一个非持续性的注释以说明它正在抖动；
- 给服务与主机相关的联系人发送一个“开始抖动”的通知；
- 压制主机与服务其他通知(这个在 [通知逻辑](#) 中有一个过滤)；

当服务或主机停止抖动时，Nagios 将会：

- 记录下主机与服务停止了抖动；
- 删除最初的给主机与服务增加的开始抖动的注释；
- 给主机与服务相关的联系人送出一个“抖动停止”的通知；
- 停止阻塞该主机与服务的通知(通知转回到正常的 [通知逻辑](#))。

8.7.9. 使能抖动感知功能

在 Nagios 打开抖动感知功能，需要如下设置：

- 将 [enable_flap_detection](#) 域设置为 1；
- 在主机与服务对象定义中的 **flap_detection_enabled** 域设置为 1；

如果想关闭全局的抖动感知功能，将 [enable_flap_detection](#) 域设置为 0；

如果只想关闭一部分主机与服务的抖动检查，使用在主机与服务对象定义里

`flap_detection_enabled` 域来控制它；

8.8. 服务和主机的定期检测

8.8.1. 未完成

本文档因 Nagios3.x 版本要重写，等后续版本再补充完善本文档...

`service_inter_check_delay`

`service_interleaving`

`max_concurrent_checks`

`host_inter_check_delay`

8.9. 有关通知的对象扩展

8.9.1. 介绍



Nagios支持对主机与服务所对应联系人通知的对象扩展。主机与服务中有关通知的对象扩展是由 [对象定义文件](#)里的 [主机扩展对象](#)和 [服务扩展对象](#)来声明的。

注意



下面例子里只给出了服务扩展对象定义，其实主机扩展对象定义也是一样的，当然，主机扩展是给主机对象的，而服务扩展只给服务对象。 :-)

8.9.2. 什么时候做通知扩展？

通知扩展将会且仅会在一个或多个扩展对象与当前要送出的通知相匹配时才做。如果主机与服务的通知与对象扩展不匹配任何一个合法的对象扩展，不会有主机或服务的对象扩展被应用于当前的通知过程中。见下面的例子：

```
define serviceescalation{  
    host_name                webserver  
  
    service_description      HTTP  
  
    first_notification        3  
  
    last_notification 5
```

```

        notification_interval    90
        contact_groups           nt-admins, managers
    }
    define serviceescalation{
        host_name                 webserver
        service_description       HTTP
        first_notification         6
        last_notification 10
        notification_interval      60
        contact_groups            nt-admins, managers, everyone
    }

```

要注意有一个通知的对象扩展定义的“孔洞”（空白区间）。也就是第 1 与第 2 个通知不会被扩展对象处理，对于超出 10 的通知也不会处理。对于第 1 和第 2 次通知，与全部的通知一样将使用服务对象里的**默认**联系人组里的联系人做对象通知。在例子中，假定服务对象定义里的默认的联系人组是名为 **nt-admins** 的联系人组。

8.9.3. 联系人组

当定义了通知相关的对象扩展，很重要的一点是要记得“低级别”对象扩展里的联系人组一定要出现在“高级别”对象扩展里的联系人组。这样才会确保每一个将要收到故障通知的人在故障不断扩张的情况下会**持续地**收到通知。例如：

```

define serviceescalation{
    host_name                 webserver
    service_description       HTTP
    first_notification         3
    last_notification 5
    notification_interval      90
    contact_groups            nt-admins, managers
}

define serviceescalation{
    host_name                 webserver
    service_description       HTTP

```

```

    first_notification      6
    last_notification 0
    notification_interval   60
    contact_groups          nt-admins, managers, everyone
}

```

第一个(“低级别”)档次的扩展包括了 **nt-admins** 和 **managers** 两个联系人组。后一个(“高级别”)档次的扩展包括了 **nt-admins**、**managers** 和 **everyone** 等三个联系人组。注意, **nt-admins** 这个联系人组被包含在两个档次的扩展里, 这样做可以使这个联系人组的成员可以在前两个通知送达后仍旧可以接到后序的通知。**managers** 联系人组最初是在第一个档次(“低级别”)的扩展里出现一里面的成员会在第三个通知开始送出时收到通知。肯定是希望 **managers** 组里的联系人可持续地收到之后的通知(如果第 5 次故障通知还在的话), 因而这个组也加到了第 2(“高级别”)档次的扩展定义里了。

8.9.4. 扩展范围的覆盖

关于通知的对象扩展可以被覆盖, 见下面的例子:

```

define serviceescalation{
    host_name            webserver
    service_description  HTTP
    first_notification    3
    last_notification 5
    notification_interval 20
    contact_groups        nt-admins, managers
}

define serviceescalation{
    host_name            webserver
    service_description  HTTP
    first_notification    4
    last_notification 0
    notification_interval 30
    contact_groups        on-call-support
}

```

在上例中,

- **nt-admins** 和 **managers** 两个联系人组将在第 3 次通知开始时收到通知；
- 全部的三个联系人组将在第 4 和第 5 次通知时收到通知；
- 仅仅是 **on-call-support** 联系人组会在第 6 次及之后的通知送出时收到通知。

8.9.5. 恢复的通知

当通知被扩展的时候，恢复通知会因故障通知状态不同而稍有不同，见下例：

```
define serviceescalation{
    host_name            webserver
    service_description  HTTP
    first_notification    3
    last_notification    5
    notification_interval 20
    contact_groups       nt-admins, managers
}

define serviceescalation{
    host_name            webserver
    service_description  HTTP
    first_notification    4
    last_notification    0
    notification_interval 30
    contact_groups       on-call-support
}
```

如果在第 3 次故障通知之后服务检测后要送出一个恢复通知，那么谁会收到通知？事实上，这个恢复通知应该算是第 4 个通知，然而 Nagios 的通知扩展代码会“聪明地判断出”其实只有收到第 3 次通知的联系人组才应该收到这个恢复通知。这时，**nt-admins** 和 **managers** 联系人组将收到这个恢复通知。（译者注：那个 on-call-support 组里的联系人不会收到！）

8.9.6. 通知间隔

还可以修改对指定主机与服务通知的送出频度，用主机扩展与服务扩展对象定义里的 **notification_interval** 域来指定不同的频度。如下例：

```
define serviceescalation{
    host_name            webserver
```

```

        service_description      HTTP
        first_notification        3
        last_notification 5
        notification_interval     45
        contact_groups            nt-admins, managers
    }

    define serviceescalation{
        host_name                  webserver
        service_description        HTTP
        first_notification          6
        last_notification 0
        notification_interval      60
        contact_groups             nt-admins, managers, everyone
    }

```

这个例子中，这个服务的默认通知送出间隔是 240 分钟(该值是在服务对象定义里设置的)。当该服务的通知被扩展到第 3、第 4 和第 5 次时，每次通知的间隔将是 45 分钟。在第 6 次及之后，通知间隔将变成 60 分钟，这个是在第 2 个的服务扩展对象里定义的。

既然主机与服务的对象扩展有可能覆盖，而且某个主机事实上有可能从属于多个主机组，那么 Nagios 就不得不就在通知间隔有覆盖的情况下取哪个通知间隔做个决定。当对于一个服务通知存在有多个合法有效的对象扩展定义时，Nagios 将会取其中最小的通知间隔来做为间隔。见下例：

```

define serviceescalation{
    host_name                  webserver
    service_description        HTTP
    first_notification          3
    last_notification 5
    notification_interval      45
    contact_groups             nt-admins, managers
}

define serviceescalation{
    host_name                  webserver

```

```

        service_description      HTTP
        first_notification        4
        last_notification 0
        notification_interval     60
        contact_groups            nt-admins, managers, everyone
    }

```

该例中有针对第 4 和第 5 次通知，有两个对象扩展相互覆盖。这两次通知间隔里，Nagios 的通知间隔将是 45 分钟，因为当这几次通知要送出时在现有的合法有效的服务对象扩展里这个值最小。

```

define serviceescalation{
    host_name            webserver
    service_description   HTTP
    first_notification    3
    last_notification 5
    notification_interval 45
    contact_groups        nt-admins, managers
}

define serviceescalation{
    host_name            webserver
    service_description   HTTP
    first_notification    4
    last_notification 6
    notification_interval 0
    contact_groups        nt-admins, managers, everyone
}

define serviceescalation{
    host_name            webserver
    service_description   HTTP
    first_notification    7
    last_notification 0
    notification_interval 30
}

```

```
contact_groups          nt-admins, managers
}
```

在上例中，故障通知的最大次数是在 4。这是因为第二档次的服务对象扩展里的通知间隔值是 0，因而(当第 4 次通知将要被送出时)只会送出一个通知而之后通知被抑制。因此，在第 4 次通知送出后第三个服务扩展对象无论如何也不会起作用了。

8.9.7. 时间周期的限制

通常的情况下，对通知的对象扩展可以用于任意想要送出主机与服务通知的时刻。这个“通知时间窗口”取决于 [主机](#) 与 [服务](#) 对象定义里的 `notification_period` 域值。

可以用主机扩展与对象扩展里的 `escalation_period` 域来指定一个特定时间周期使得扩展被限定只处于某个特定时间段内。使用 `escalation_period` 域来指定某个 [时间周期](#) 里对象扩展是可用的，对象扩展将只是在指定的时间里可用。如果没有在 `escalation_period` 域里指定时间周期，主机扩展与服务扩展将会在“通知时间窗口”内的任意时间里是可用的。

注意



通知扩展依旧会受限于主机与服务对象定义里的 `notification_period` 域所指定的时间周期，因而特定的对象扩展里的时间周期是一个更大范围“通知时间窗口”的子集。

8.9.8. 状态限制

如果想只是想用特定的主机与服务状态限定针对通知的扩展，可以用主机扩展和服务扩展对象里的 `escalation_options` 域来指定。如果没有指定 `escalation_options` 域，针对通知的扩展将作用于主机与服务的任何状态之上。

8.10. 应召循环

8.10.1. 介绍

(原文档题目使用的是 On-Call，有“随叫随到、应召”的意思，所以翻译为“应召”——译者注)



管理员通常要承担着响应 PB 机呼叫、电话等工作，即使他们不情愿。没人喜欢在清晨四点被叫起处理问题，但午夜修正错误会肯定会好些，总比在早上九点闲庭信步的时候碰到一脸怒气的老板要强许多吧。

应召循环设定好后给那些承担起应急响应的一小搓倒霉蛋。通常在周末、晚间和假日里，有多个管理员轮流值守。

下面会以更容易地完成应召循环的方式来创建 [时间周期](#) 对象定义，这些定义无法处理人为因素(如管理员生病、换岗或是手机没电等)，但可以建立一个基础计划框架在大部分时间里它肯定是适用的。

8.10.2. 场景 1：假日与周末

John 和 Bob 是两个管理员负责对 Nagios 报警做出响应。John 每周工作时段内(不包括假日)负责接收全部通知，由 Bob 负责在每周末和假日里接收通知，Bob 是那个幸运的倒霉蛋。下面给出如何定义时间周期对象定义...

首先，定义一个假日的时间周期对象：

```
define timeperiod{
    name      holidays
    timeperiod_name  holidays

    january 1          00:00-24:00      ; New Year's Day
    2007-03-23          00:00-24:00      ; Easter (2008)
    2007-04-12          00:00-24:00      ; Easter (2009)
    monday -1 may       00:00-24:00      ; Memorial Day (Last Monday in May)
    july 4              00:00-24:00      ; Independence Day
    monday 1 september  00:00-24:00      ; Labor Day (1st Monday in
September)
    thursday 4 november 00:00-24:00      ; Thanksgiving (4th Thursday in November)
    december 25          00:00-24:00      ; Christmas
    december 31          17:00-24:00      ; New Year's Eve (5pm onwards)
}
```

下一步，给 John 定义一个应召时间周期对象，是平时每周的早晚工作时间，但不包括假日时段：

```
define timeperiod{
    timeperiod_name      john-oncall

    monday               00:00-24:00
    tuesday               00:00-24:00
```

```
wednesday          00:00-24:00

thursday          00:00-24:00

friday            00:00-24:00

exclude           holidays      ; Exclude holiday dates/times defined elsewhere
}
```

下面可以在 John 的联系人定义里引用这个时间周期定义：

```
define contact{

    contact_name      john

    ...

    host_notification_period      john-oncall

    service_notification_period   john-oncall

}
```

给 Bob 定义一个应召时间段，包括每个周末和假日：

```
define timeperiod{

    timeperiod_name      bob-oncall

    friday              00:00-24:00

    saturday            00:00-24:00

    use                 holidays ; Also include holiday date/times defined elsewhere

}
```

在 Bob 的联系人定义里引用这个时间周期定义：

```
define contact{

    contact_name      bob

    ...

    host_notification_period      bob-oncall

    service_notification_period   bob-oncall

}
```

8.10.3. 场景 2：轮换值班

在该场景里 John 和 Bob 轮换地交接值班，不管这一天是周末、工作日还是假日。

定义一个时间周期对象，在该时间内由 John 负责接收通知，隔天工作。假定开始这天是 2007 年 8 月 1 日的全天，这个对象定义是：

```
define timeperiod{
    timeperiod_name      john-oncall
    2007-08-01 / 2      00:00-24:00      ; Every two days, starting August 1st, 2007
}
```

定义一个时间周期该由 Bob 负责接收通知，Bob 接收通知的时候 John 不工作，所以他起始时间是 2007 年 8 月 2 日开始。

```
define timeperiod{
    timeperiod_name      bob-oncall
    2007-08-02 / 2      00:00-24:00      ; Every two days, starting August 2nd, 2007
}
```

下面可以在 John 和 Bob 的联系人对象定义里引用这两个时间周期定义：

```
define contact{
    contact_name          john
    ...
    host_notification_period      john-oncall
    service_notification_period   john-oncall
}

define contact{
    contact_name          bob
    ...
    host_notification_period      bob-oncall
    service_notification_period   bob-oncall
}
```

8.10.4. 场景 3：轮换周值班

在这个场景里，由 John 和 Bob 每周一个地轮换周值班。由 John 负责干一周，Bob 再干一周，周而复始。

定义 John 需要接收通知的时间周期对象，假定开始这一天是 2007 年 7 月 29 日开始的一周，定义如下：

```
define timeperiod{
    timeperiod_name      john-oncall
    2007-07-29 / 14  00:00-24:00      ; Every 14 days (two weeks), starting Sunday, July
29th, 2007
    2007-07-30 / 14  00:00-24:00      ; Every other Monday starting July 30th, 2007
    2007-07-31 / 14  00:00-24:00      ; Every other Tuesday starting July 31st, 2007
    2007-08-01 / 14  00:00-24:00      ; Every other Wednesday starting August 1st, 2007
    2007-08-02 / 14  00:00-24:00      ; Every other Thursday starting August 2nd, 2007
    2007-08-03 / 14  00:00-24:00      ; Every other Friday starting August 3rd, 2007
    2007-08-04 / 14  00:00-24:00      ; Every other Saturday starting August 4th, 2007
}
```

给 Bob 的工作时间定义一个时间周期对象定义。Bob 是在 John 干完后的一周开始做应召值班，因而开始时间是 2007 年 8 月 5 日。

```
define timeperiod{
    timeperiod_name      bob-oncall
    2007-08-05 / 14  00:00-24:00      ; Every 14 days (two weeks), starting Sunday, August
5th, 2007
    2007-08-06 / 14  00:00-24:00      ; Every other Monday starting August 6th, 2007
    2007-08-07 / 14  00:00-24:00      ; Every other Tuesday starting August 7th, 2007
    2007-08-08 / 14  00:00-24:00      ; Every other Wednesday starting August 8th, 2007
    2007-08-09 / 14  00:00-24:00      ; Every other Thursday starting August 9th, 2007
    2007-08-10 / 14  00:00-24:00      ; Every other Friday starting August 10th, 2007
    2007-08-11 / 14  00:00-24:00      ; Every other Saturday starting August 11th, 2007
}
```

现在可以在 John 和 Bob 的联系人对象定义里引用这两个时间周期定义了：

```
define contact{
    contact_name          john
    ...
```

```

        host_notification_period      john-oncall
        service_notification_period    john-oncall
    }
define contact{
    contact_name      bob
    ...
    host_notification_period      bob-oncall
    service_notification_period    bob-oncall
}

```

8.10.5. 场景 4: 假期

在该场景里，John 在全部时间内处理报警，除非他休假。每个月他会固定休假同时还有一个长假，在 John 休假时由 Bob 来负责处理报警。

先给 John 定义一个休假和长假的时间段：

```

define timeperiod{
    name      john-out-of-office
    timeperiod_name  john-out-of-office
    day 15          00:00-24:00          ; 15th day of each month
    day -1          00:00-24:00          ; Last day of each month (28th,
29th, 30th, or 31st)
    day -2          00:00-24:00          ; 2nd to last day of each month
(27th, 28th, 29th, or 30th)
    january 2              00:00-24:00          ; January 2nd each year
    june 1 - july 5        00:00-24:00          ; Yearly camping trip (June 1st
- July 5th)
    2007-11-01 - 2007-11-10  00:00-24:00          ; Vacation to the US Virgin
Islands (November 1st-10th, 2007)
}

```

再定义出 John 的日常的应召时间安排，不包括上面的 John 的假日：

```

define timeperiod{
    timeperiod_name      john-oncall

```

```

monday          00:00-24:00
tuesday         00:00-24:00
wednesday              00:00-24:00
thursday         00:00-24:00
friday           00:00-24:00
exclude          john-out-of-office          ; Exclude dates/times John is out
}

```

下面可以在 John 的联系人对象里引用这个时间周期定义：

```

define contact{
    contact_name          john
    ...
    host_notification_period      john-oncall
    service_notification_period    john-oncall
}

```

再给 Bob 定义一个时间段，内容是 John 的节假日：

```

define timeperiod{
    timeperiod_name      bob-oncall
    use                   john-out-of-office          ; Include holiday date/times that John is
out
}

```

下面就可以在 Bob 的联系人对象里引用这个时间周期定义：

```

define contact{
    contact_name          bob
    ...
    host_notification_period      bob-oncall
    service_notification_period    bob-oncall
}

```

8.10.6. 其他场景

可能会有各种各样的其他应召工作场景。在 [时间周期对象定义](#) 的日期例外项将可以处理几乎全部的日期相关或日期段相关的工作时间定义，请审视一下可用的不同时间格式。如果创建的时间段定义有错误，其结果是某个人总会是不能在指定时间内工作的。:-)

8.11. 主机间与服务间依赖关系

8.11.1. 介绍

主机与服务的依赖是 Nagios 的高级特性，它可用于基于一个或多个其他主机与服务来控制当前主机与服务的行为。下面将解释一下依赖关系是如何工作的，包括主机间的和服务间的依赖差异。

8.11.2. 服务依赖概况

服务依赖的几个基本点：

- 服务可以依赖于一个或多个其他服务；
- 服务可以依赖于绑定于不同主机上的服务；
- 服务依赖是不被继承的(除非专门配置过)；
- 服务依赖可被用于在不同的状态情况(正常、告警、未知和紧急)下引发服务检测的执行和服务通知的抑制；
- 服务依赖可能只是在指定 [时间周期](#) 内合法。

8.11.3. 定义服务依赖

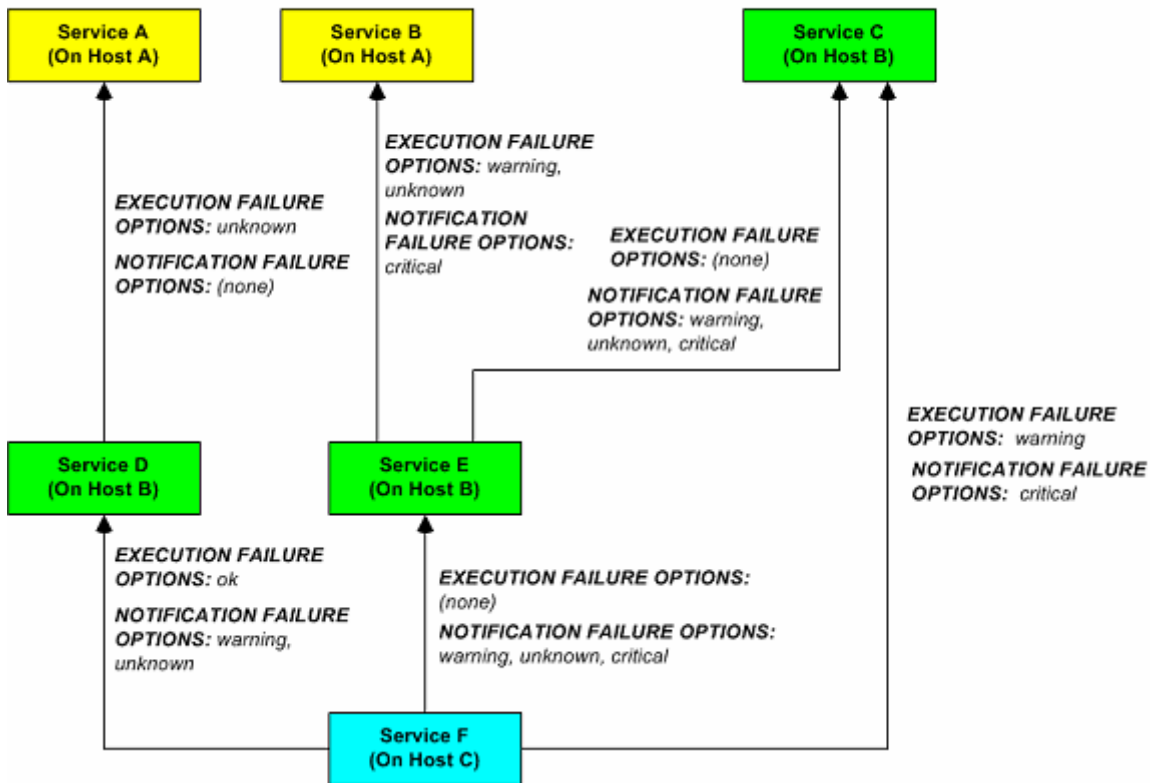
首先做为基础。应在 [对象配置文件](#) 里创建 [服务依赖对象定义](#)。每个服务依赖定义要指定 **依赖于** 哪个服务，作为 **被依赖** 的服务的选取标准是当其失效时会引发执行与通知动作(下面会解释)。

可以给一个服务创建多个服务依赖，但必须要给每个依赖创建各自独立的依赖对象。

8.11.4. 服务依赖对象的样例

下图中给出一个服务通知与执行依赖的逻辑示意，不同服务依赖于其他服务的通知和检测执行。

Service Dependencies



在这个例子中，在 Host C 主机上的 Service F 的服务依赖将被定义成这样：

```

define servicedependency{
    host_name                Host B
    service_description       Service D
    dependent_host_name       Host C
    dependent_service_description Service F
    execution_failure_criteria
    notification_failure_criteria w,u
}

define servicedependency{
    host_name                Host B
    service_description       Service E
    dependent_host_name       Host C
    dependent_service_description Service F
    execution_failure_criteria
    notification_failure_criteria w,u,c
}
  
```



```

}

define servicedependency{

    host_name                Host B

    service_description       Service C

    dependent_host_name       Host C

    dependent_service_description    Service F

    execution_failure_criteria

    notification_failure_criteria    c

}

```

在图中的其他服务依赖将被定义成这样：

```

define servicedependency{

    host_name                Host A

    service_description       Service A

    dependent_host_name       Host B

    dependent_service_description    Service D

    execution_failure_criteria

    notification_failure_criteria    n

}

define servicedependency{

    host_name                Host A

    service_description       Service B

    dependent_host_name       Host B

    dependent_service_description    Service E

    execution_failure_criteria    w,u

    notification_failure_criteria    c

}

define servicedependency{

    host_name                Host B

    service_description       Service C

    dependent_host_name       Host B

```

```

    dependent_service_description      Service E

    execution_failure_criterion

    notification_failure_criteria      w, u, c
}

```

8.11.5. 如何测试服务依赖？

在 Nagios 进行一个服务的检测或是送出该服务的通知之前，将会查看该服务是否有服务依赖。如果没有，那么象正常情况一样做检测或送出服务通知。如果该服务**存在**一个或多个服务依赖，Nagios 将会如下方式来检查每个服务依赖：

- Nagios将取出给定的当前 ***服务依赖** 的服务状况；
- Nagios 用当前有服务依赖的服务状态去比对依赖对象定义(里面有关时间的设置)里所给出的执行或通知失效的选项；
- 如果当前有服务依赖的服务状态匹配中其中一个失效选项，依赖就失效并会中断依赖检测的逻辑循环；
- 如果当前有服务依赖的服务状态没有匹配中任何一个失效选项，依赖检查通过并且 Nagios 将继续运行并检查下一个依赖入口；

这个检测循环会继续直到全部的服务依赖都检查完成或是其中一个服务依赖的失效选项被命中。

注意



注：*重要的是，默认情况下，Nagios在进行依赖检查时将会使用该服务的最近的 [硬态](#) 状态所匹配的服务依赖。如果想让Nagios使用最近的状态(不管是软态状态还是硬态状态)来做服务的依赖匹配，需要使能 [soft state dependencies](#) 选项。

8.11.6. 实施依赖

当服务的 [主动检测](#) 将要被执行时可以用实施服务依赖来限制它，[强制检测](#) 并不会被实施服务依赖所限制。

如果针对该服务依赖的**全部**测试都**通过**，Nagios将会象一般情况一样来执行针对该服务的检测。如果即使只有一个针对该服务依赖的测试没有通过，Nagios也将临时阻止针对该服务的检测，而在之后可能会通过针对该服务依赖的全部测试。如果是这样的话，Nagios将会再次来执行针对该服务的检测。更多的关有计划检测逻辑信息可以查阅 [这篇文档](#)。

在上例中，服务 E 将在服务依赖检测中因为服务 B 处于告警或未知状态时测试失败。如果是这样的话，服务 E 的检测将不会执行而在此之后到计划检测时将可能再做。

8.11.7. 通知依赖

如果针对该服务的**全部**通知依赖检测都**通过**，Nagios将会象一般情况一样送出该服务的通知。如果即便只有一个针对该服务的通知依赖没有通过测试，Nagios也将临时阻止送出针对该服务的通知，而之后可能会有针对该服务的通知依赖检测全部通过。如果是这样的话，Nagios将再次执行针对该服务的检测，更多的有关通知逻辑信息可以查阅 [这篇文档](#)。

在上例中，服务 F 将在通知依赖检测中因服务 C 处于紧急状态而测试失败，**也可能因**服务 D 处于告警或未知状态，**也可能因**服务 E 处于告警或未知或紧急状态，也会使测试失败。如果是这样的话，将不会送出针对该服务的通知。

8.11.8. 依赖关系的继承

前面已经讲过，服务依赖关系默认是**不会被继承**的。在例子中，可以看到服务 F 依赖于服务 E，然而，它并不会从服务 E 的依赖定义里继承对服务 B 和服务 C 的依赖关系。为使服务 F 依赖于服务 C 必须加入另一个服务依赖对象定义，而因为没有对服务 B 的依赖关系定义，因而服务 F 是**不会**依赖于服务 B 的。

如果**真的希望**让服务依赖关系继承，必须用 [服务依赖关系](#)对象定义里的**inherits_parent**域来标识说明。当这个域使能时，说明该服务依赖继承了**来自指向源服务的服务依赖关系**(就是父节点服务依赖于什么服务它也一样要依赖于那些服务)，也就是说如果源服务依赖的服务有一个依赖关系的检测失败的话，这个服务依赖的检测也会失败。

在上例中，设想一下，如果要加入一个针对服务 F 的新服务依赖关系定义使之依赖于服务 A，可以给服务 F 创建一个新的服务依赖关系使之**依赖于**服务 A，并让服务 A 做**服务依赖源**(也就是这个服务状态**决定其后服务**)。还可以修改针对服务 D 与服务 F 的服务依赖对象定义，象这样：

```
define servicedependency{
    host_name                Host B
    service_description      Service D
    dependent_host_name       Host C
    dependent_service_description  Service F
    execution_failure_criteria
    notification_failure_criteria    n
    inherits_parent           1
}
```

因为 **inherits_parent** 域使能了，那么当测试服务 F 对服务 D 的依赖时也会和服务 D 与服务 A 的之间的依赖情况进行测试。

依赖关系可以在多种层次上继承，如果服务 D 与服务 A 之间的服务依赖关系对象定义里的 `inherits_parent` 域也使能(设置为 1)时，并且服务 A 依赖于某个其他服务(比如说是服务 G)，那么服务 F 依赖于服务 D、服务 A 和服务 G(但每层依赖关系的要求是不一样的)。

8.11.9. 主机依赖

正如期待的那样，主机的依赖关系也跟服务依赖关系一样的方式实现，只是针对的是主机间的关系而不服服务间的关系。

提示



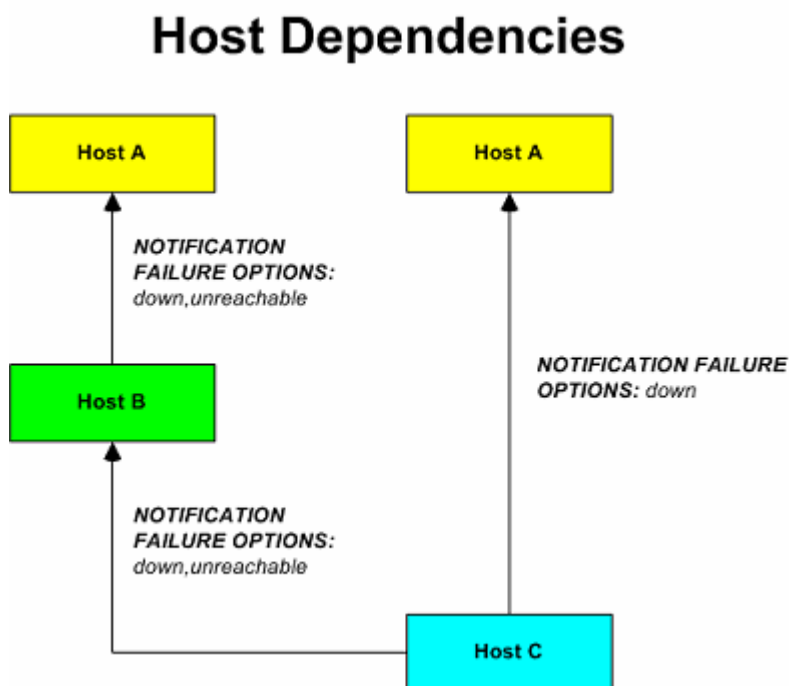
不要把主机间依赖关系与主机的节点父子关系混淆。很多情况下，可以用主机的节点父子关系(在 [主机对象](#) 定义里的 `parents` 域来定义这种关系)而不是主机间的依赖关系来表达。有关主机父子关系如何工作的说明可以在 [网络可达性](#) 这篇文档中找到。

这有几个有关主机依赖的基本概念：

- 一个主机可以依赖于一个或多个其他主机；
- 主机依赖关系默认是不被继承的(除非专门配置声明)；
- 主机依赖关系可被用于在不同状态环境(运行、宕机或不可达等)时抑制主机检测和主机通知；
- 主机依赖关系只是在设定的 [时间周期](#) 区间段内是合法有效的。

8.11.10. 主机依赖关系的样例

下图示意了一个主机通知依赖关系的逻辑拓扑。在通知时不同的主机依赖于其他的主机。



在上图例子中，针对主机 C 的依赖关系定义将会是这样的：

```

define hostdependency{
    host_name            Host A
    dependent_host_name   Host C
    notification_failure_criteria    d
}

define hostdependency{
    host_name            Host B
    dependent_host_name   Host C
    notification_failure_criteria    d,u
}

```

象服务依赖关系定义一样，主机依赖不会继承。在上图中，主机 C 并没有继承来自主机 B 的主机依赖关系。为使主机 C 也依赖于主机 A，必须给它创建一个新的主机依赖关系定义。

主机通知的依赖关系处理机制与服务通知的依赖关系处理机制相似。如果对该主机的**全部**通知依赖关系的测试都**通过**的话，Nagios将象一般情况那样送出该主机的通知。如果即便是有一个该主机的通知依赖关系没有测试通过，Nagios也将会临时阻止针对该主机的通知送出。在此之后可能针对该主机的通知依赖关系全部通过，如果这种情况发生，Nagios将会象一般情况那样再送出该主机的通知。更多关有通知逻辑的信息可以查阅 [这篇文档](#)。

8.12. 依赖检测的前处理

8.12.1. 介绍

主机和服务的 [依赖](#)关系（从属关系、上下级关系）的定义可令你在执行检测时和在进行告警送出时拥有更大的控制力。一旦在监控过程中运用了关系定义，非常重要确保在依赖关系逻辑之上的状态信息保持同步，越新越好。

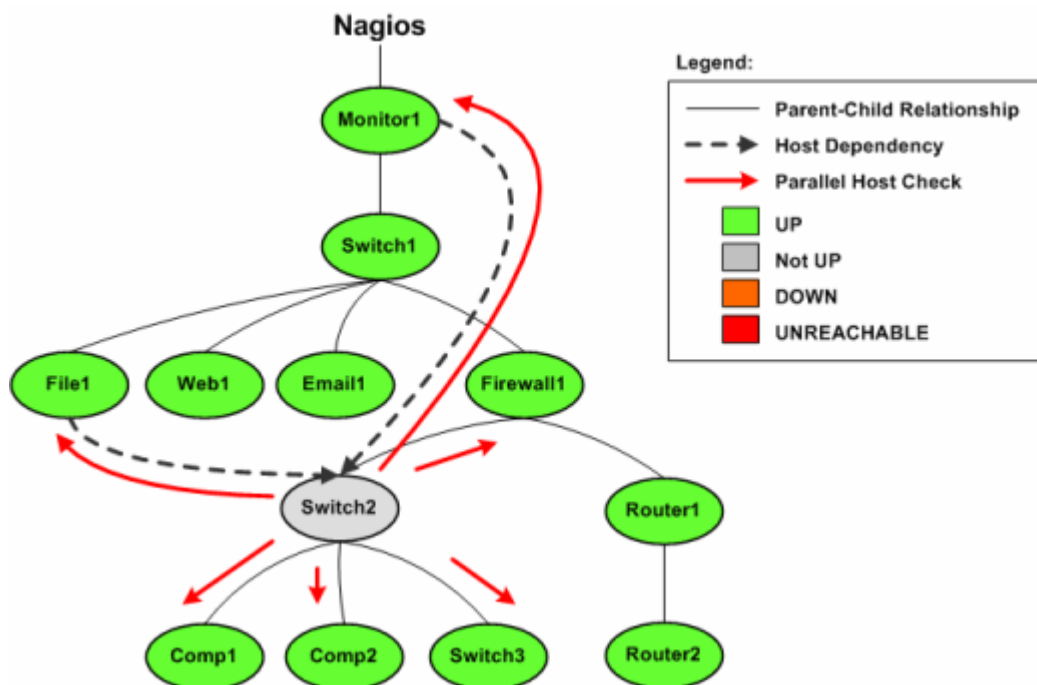
在它决定是否要送出报警或是允许对主机或服务进行自主检测时，Nagios 允许你在进行针对主机和服务的依赖检测前做些准备以确认依赖逻辑将是最新的状态信息。

8.12.2. 如何进行依赖检测前准备工作？

下图示意了一个被 Nagios 监控的主机组图，包含它们的父子节点关系及依赖关系定义。

图例中的**Switch2** 主机刚好从运行状态到出问题的状态。Nagios需要判断主机是否是宕机或是不可达，因而它会运行并行检测针对**Switch2** 的直接父节点 (**Firewall1**)和子节点 (**Comp1**、**Comp2** 和**Switch3**)。这个是 [主机可达性](#)检查函数的一般逻辑。

你或许注意到了 **Switch2** 是依赖于 **Monitor1** 和 **File1** 以进行告警和执行检测(这点在本例中并不重要)。如果主机依赖检测准备使能的话, Nagios 将会在针对 **Switch2** 的直接父节点检测的同时针对 **Monitor1** 和 **File1** 进行并行检测。Nagios 这样做是因为很快就必须进行依赖逻辑检查(例如需要告警)并且将要确保在依赖关系逻辑之中的与主机关系的部分的信息是最新的。



这就是进行的依赖检测前准备工作, 很简单, 不是么?

注意



服务依赖检测前的准备工作与之类似, 只不过是把针对主机替换成针对服务。

8.12.3. 使能检查准备

依赖检测的准备涉及上面很少的部分, 所以我推荐你打开这个功能。在许多情况下, 拥有在依赖逻辑上的准确状态信息比过多地进行检测更具意义。

使能依赖检测准备很简单:

- 针对主机的依赖检测准备由 [enable_predictive_host_dependency_checks](#) 选项控制。
- 针对服务的依赖检测准备由 [enable_predictive_service_dependency_checks](#) 选项控制。

8.12.4. 缓存检测

依赖检测准备是一种按需生成的检测方式且服从 [缓存检测](#) 的规则。缓存检测让Nagios提供性能提升, 主要是利用与这些主机和服务相关的最近检测结果替代对实际主机和服务的检测。更多关于缓存检测的内容可在 [这里](#) 找到。

8.13. 性能数据

8.13.1. 介绍

Nagios设计成可以处理 [插件](#)检测返回状态数据的同时可选地做性能数据处理，也就是说，可由外部应用来处理性能数据。下面说明一下不同性能数据类型以及这些信息是如何被处理的...

8.13.2. 性能数据的类型

在 Nagios 里可以有两大类性能数据：

- 检测过程的性能数据
- 插件返回的性能数据

检测过程的性能数据是与主机检测和服务检测相关的系统内部数据。这些数据包括象服务检测延时(就是检测实际检测的时刻与其计划时间之间的推后时间)和主机检测与服务检测执行所花费的时间。这类性能数据对全部可执行的检测都适用。[\\$HOSTEXECUTIONTIME\\$](#)和 [\\$SERVICEEXECUTIONTIME\\$宏定义](#)可用于度量主机或服务检测所运行的时间，[\\$HOSTLATENCY\\$](#)和 [\\$SERVICELATENCY\\$](#)宏定义可用于度量主机和服务检测的执行延时。

插件返回的性能数据是由主机与服务检测时插件检测结果带出来的外部数据。特定插件的数据，象丢包率、磁盘空闲空间、处理器负荷、当前登录的用户数等，是在执行时由插件自己来测量出来的任何一种类型数据。特定插件数据是可选项并非每个插件都有。(如果有)特定插件数据将包含在 [\\$HOSTPERFDATA\\$](#)和 [\\$SERVICEPERFDATA\\$宏定义](#)里。更多有关如何在Nagios里返回性能数据蕴涵在[\\$HOSTPERFDATA\\$](#)和 [\\$SERVICEPERFDATA\\$](#)宏里面。

8.13.3. 插件返回的性能数据

最小情况时，Nagios 插件须用一行可读字符串来带出状态和相关测试值。例如，check_ping 插件可以返回象这样的一行内容：

```
PING ok - Packet loss = 0%, RTA = 0.80 ms
```

在这种简单输出内容里，整行内容都包含于[\\$HOSTOUTPUT\\$](#)或[\\$SERVICEOUTPUT\\$宏](#)里面(取决于这个插件是被用于主机检测还是被用于服务检测)。

插件一般返回性能数据的方式是在插件可读的输出行里加上管道符(|)，后面跟一个或几个性能测量值。还是用 check_ping 插件做例子，假定用这种方式来返回性能数据，其插件输出内容象是这样：

```
PING ok - Packet loss = 0%, RTA = 0.80 ms | percent_packet_loss=0, rta=0.80
```

当 Nagios 处理象这样的输出内容格式时，将把它分为两部分：

- 管道符之前的内容被认为是“正常”的插件输出并保存于[\\$HOSTOUTPUT\\$](#)或是[\\$SERVICEOUTPUT\\$](#)宏里；
- 管道符之后的内容被认为是与性能数据相关的内容并保存于[\\$HOSTPERFDATA\\$](#)或是 [\\$SERVICEPERFDATA\\$](#)宏里；

在上例中，`$HOSTOUTPUT$`或`$SERVICEOUTPUT$`宏的内容将是“PING ok - Packet loss = 0%, RTA = 0.80 ms”（没有引号），而`$HOSTPERFDATA$`或`$SERVICEPERFDATA$`宏将是“percent_packet_loss=0, rta=0.80”（没有引号）。

插件输出可以包含有多行的性能数据输出（也象正常正文输出），这个在 [插件API文档](#) 有介绍。

注意



Nagios 的守护程序并不直接处理插件的性能数据，因而它并不关心在性能数据里有什么东西。对性能数据并没有什么限制与格式约束，但是，如果要用外部构件来处理性能数据（如 PerfParse），外部构件需要插件以固定格式来输出性能数据。这要审视将要使用的外部构件里相关文档。

8.13.4. 性能数据的处理

如果想要 Nagios 和插件生成并处理性能数据，需要按下面来做：

- 打开 [process_performance_data](#) 选项开关；
- 配置 Nagios 以便于把性能数据要么写入文件要么执行一个数据处理命令；

查阅文档看如何把性能数据写入文件或是如何来执行数据处理命令；

8.13.5. 用命令来处理性能数据

在Nagios里最柔性化地处理性能数据的方式是使用命令来处理性能数据或是把数据重定向以便于外部应用来做后序处理。在Nagios里对主机和服务性能数据的处理命令分别取决于对

[host_perfdata_command](#)和 [service_perfdata_command](#)选项设置。

下例是重定向服务检测的性能数据到一个文本文件里以让其他应用来做后序处理：

```
define command{
    command_name      store-service-perfdata
    command_line      /bin/echo -e
"$LASTSERVICECHECK$\t$HOSTNAME$\t$SERVICEDESC$\t$SERVICESTATE$\t$SERVICEATTEMPT$\t$SERVICES
TATETYPE$\t$SERVICEEXECUTIONTIME$\t$SERVICELATENCY$\t$SERVICEOUTPUT$\t$SERVICEPERFDATA$" >>
/usr/local/nagios/var/service-perfdata.dat
}
```

提示



在此方法中，虽然柔性化但也会带来 CPU 高负荷。如果想把大量的主机与服务的性能数据写入文件让外部应用来处理，可以让 Nagios 代替命令而直接写性能数据文件。这个方法在下一节里有说明。

8.13.6. 将性能数据写入文件

用 [host_perfdata_file](#) 和 [service_perfdata_file](#) 选项可以让 Nagios 直接把性能数据写入文本文件，而主机与服务性能数据的写入格式由 [host_perfdata_file_template](#) 和 [service_perfdata_file_template](#) 选项设置决定。

下例是个服务性能数据的格式设置：

```
service_perfdata_file_template=[SERVICEPERFDATA]\t$TIMET$\t$HOSTNAME$\t$SERVICEDESC$\t$SERVICEEXECUTIONTIME$\t$SERVICELATENCY$\t$SERVICEOUTPUT$\t$SERVICEPERFDATA$
```

默认情况下，文本文件以“追加”方式打开。如果想改变文件打开方式，如“写入”或是“非阻塞式读写”（用管道写文件时用的），可以设置 [host_perfdata_file_mode](#) 和 [service_perfdata_file_mode](#) 选项。

另外，还可以让 Nagios 定期地执行命令来定期处理性能数据文件（如滚动数据），用 [host_perfdata_file_processing_command](#) 和 [service_perfdata_file_processing_command](#) 选项设置。这些命令的执行间隔分别由 [host_perfdata_file_processing_interval](#) 和 [service_perfdata_file_processing_interval](#) 选项控制。

第 9 章 Nagios 专业话题

第 9 章 Nagios 专业话题

9.1. 趣事与玩笑

跟标准的监控程序不一样，Nagios 可以做些很有趣的事情。与其花费时间玩 [Quake](#)，为何不花点时间看看这个 [http://www.nagios.org/docs/hacks/...](http://www.nagios.org/docs/hacks/)

9.2. 分布式监控

9.2.1. 介绍

Nagios 可以配置为分布式监控网络服务与资源。下面将尽可能详细阐述如何实现...

9.2.2. 目标

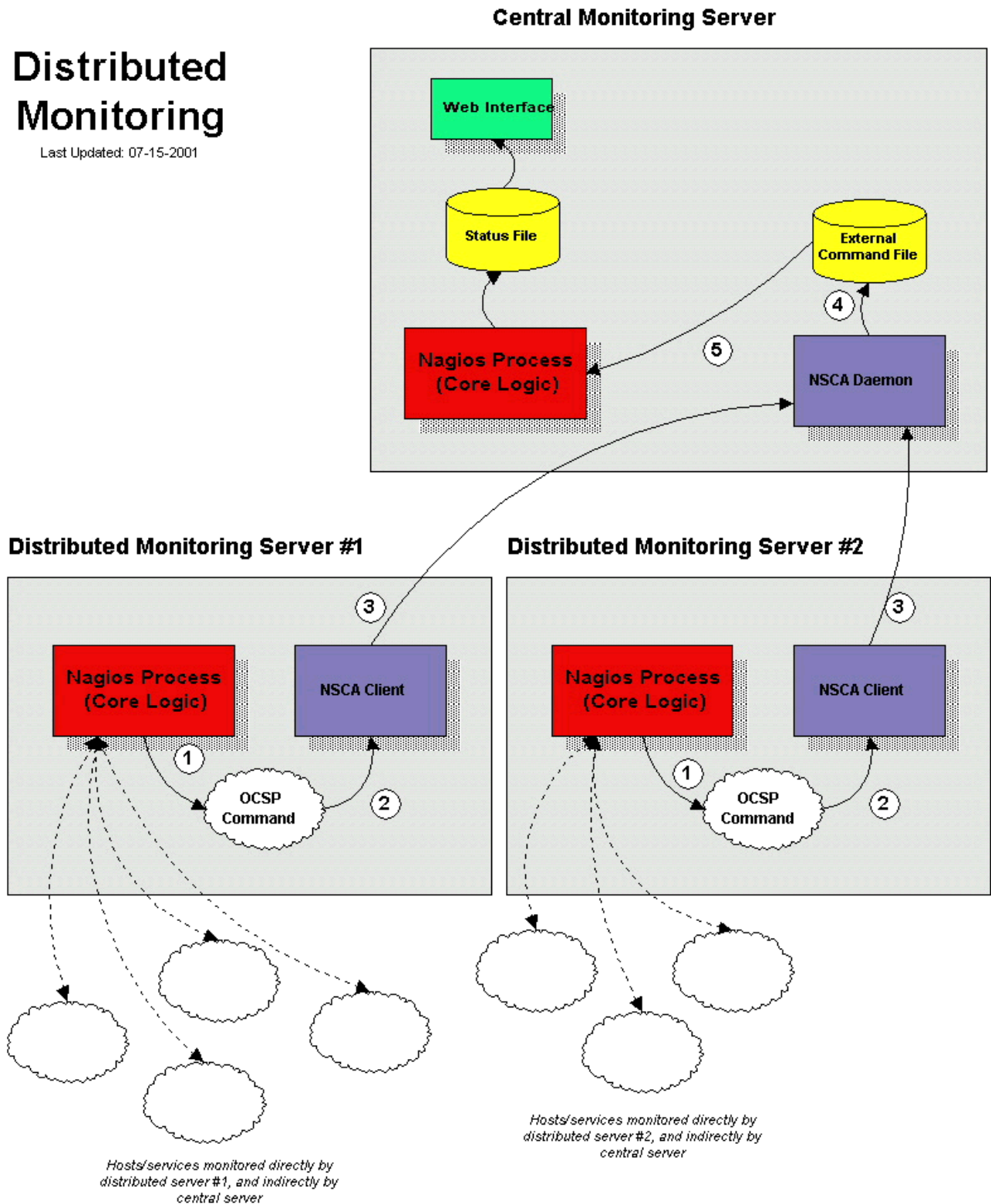
安装一个分布式监控环境的主旨是要降低整体消耗（CPU 利用率等），通过一个或多个“分布”服务器检测并将结果送给“中心”监控服务器来实现这一目的。不少小型或中型的系统将不会用到这种环境，然而，如果要 Nagios 监控上百台的主机乃至上千台主机（和几倍于它的服务个数）时，这就变得非常重要了。

9.2.3. 参照示意图

下图将帮助理解 Nagios 的分布式监控环境如何工作。将利用下图来解释一些概念...

Distributed Monitoring

Last Updated: 07-15-2001



9.2.4. 中心服务与分布服务的对比

当用 Nagios 建立一个分布监控环境时，在中心与分布服务器的配置方面有很大不同。下面将会给出两者的配置并指出在整体上这种配置的影响。先来解释一下这两种位置上不同的服务目标...

“分布”服务器的功能是真正地完成你所划分出一“组”主机的检测工作。这里的“组”定义是松散的一完成基于你的网络情况而自然形成的。在一个物理位置里可能会有若干个“组”，这取决于你的网络层次划分，

要么因为 WAN 而划分开，要么因个自独立的防火墙而划分开。很重要的一点是，在每个“组”里都只有一个运行 Nagios 的服务器并完成对该“组”的监控检测工作。分布服务器通常上面只安装有 Nagios，它不需要安装 Web 接口，如果不想让它来做也可以不送出通知、运行事件处理脚本或是执行任何其他服务检测。有关分布服务器更详尽的内容将在下面配置中给出...

“中心”服务器的目标是从一个或多个分布式服务器收集服务检测结果。虽然中心服务器偶尔也会做些自主检测，但自主检测更多只是在极端情况下才做的，因而可以说中心服务器当前只做强制检测。既然中心服务器从一台或多台分布服务器收集 [强制服务检测](#) 结果，那它就承担全部监控逻辑的整体输出工作(如送出通知、运行事件处理脚本、判定主机状态、安装并提供Web接口等)。

9.2.5. 从分布服务器上收集服务检测信息

在研讨配置细节之前，需要了解如何将分布服务器上的服务检测结果送到中心服务器。前面已经讨论过如何提交由Nagios发出的强制检测结果到该Nagios主机(在 [强制检测](#) 一文中说明)，但并没有给出任何关于提交不同主机强制检测结果的信息。

为完成从远程主机提交强制检测结果，特意编写了 [NSCA外部构件](#)。该外部构件包括两部分，第一部分是客户端程序(send_nsca)，运行于远程主机上并负责将强制检测结果送到指定的服务器上去，另一部分是NSCA守护进程(nsca)，它既可以独立地运行于守护服务也可以注册到inetd里作为一个inetd客户程序来提供监听联接。从客户端收到服务检测结果信息之后，守护进程将结果提交给在中心服务器的Nagios，方式是通过在 [外部命令文件](#)里插入一条PROCESS_SVC_CHECK_RESULT命令，之后跟上检测结果。在Nagios下一次处理 [外部命令时](#)将会找到这条由分布式服务器送来的强制检测信息并处理它。很简单对吧？

9.2.6. 分布式监控服务的配置

那么如何来配置一台分布式的 Nagios 服务器？基本要求是一个纯净安装的 Nagios，没必要安装 Web 接口或通知送出服务，因为它们都由中心服务器来完成。

配置的关键差异：

- 在分布式服务器的 [对象配置文件](#)里只定义那些由它直接监控的主机与服务的对象；
- 将分布式服务器的 [enable_notifications](#)域设置为 0，这将阻止它直接送出任何通知信息；
- 分布式服务器被配置为 [强迫型服务\(obsess over services\)](#)类型；
- 分布式服务有一个 [强迫型服务处理命令\(OCSP\)](#)的定义(下面有说明)。

为使网络内的信息充分汇总处理，需要将每个分布服务器上的**全部服务检测**结果送到中心Nagios服务器。可以用 [事件处理](#)来报告服务状态的**变换情况**，但那只是没剪裁的。为强制让分布服务器提交全部的服务检测结果，必须使能位于主配置文件里的强迫型服务 [obsess over services](#)选项并且设定好一个强迫型服务处理命令 [ocsp_command](#)以在每次服务检测完成后执行该命令。将利用强迫型服务处理命令来从分布服

务器向中心服务器送达全部服务检测结果，这中间将利用send_nsca客户端和nsca守护服务(上面已经说明)来进行数据传输过程。

为实现这一目标，需要象下面这样定义一个强迫型服务处理(ocsp)命令：

`ocsp_command=submit_check_result`

这个 `submit_check_result` 命令的定义会象是这样：

```
define command{
    command_name      submit_check_result
    command_line      /usr/local/nagios/libexec/eventhandlers/submit_check_result
$HOSTNAME$ ' $SERVICEDESC$' $SERVICESTATE$ ' $SERVICEOUTPUT$'
}
```

该 `submit_check_result` 的 SHELL 脚本的内容象是这样(用**中心服务器** IP 地址替换里面的 `central_server`):

```
#!/bin/sh

# Arguments:

# $1 = host_name (Short name of host that the service is
#      associated with)
# $2 = svc_description (Description of the service)
# $3 = state_string (A string representing the status of
#      the given service - "OK", "WARNING", "CRITICAL"
#      or "UNKNOWN")
# $4 = plugin_output (A text string that should be used
#      as the plugin output for the service checks)
#

# Convert the state string to the corresponding return code
return_code=-1

case "$3" in
    OK)
```

```

        return_code=0
        ;;
    WARNING)
        return_code=1
        ;;
    CRITICAL)
        return_code=2
        ;;
    UNKNOWN)
        return_code=-1
        ;;
esac

# pipe the service check info into the send_nscd program, which
# in turn transmits the data to the nsca daemon on the central
# monitoring server

/bin/printf "%s\t%s\t%s\t%s\n" "$1" "$2" "$return_code" "$4" |
/usr/local/nagios/bin/send_nscd central_server -c /usr/local/nagios/etc/send_nscd.cfg

```

上面脚本中假定已经有 send_nscd 客户端程序，它放在 **/usr/local/nagios/bin/** 目录里，并且把配置文件 (send_nscd.cfg) 放在 **/usr/local/nagios/etc/** 目录里。

就这么多！现在已经算是把一个远程的 Nagios 服务器配置成为一个分布服务器了。看一下在分布服务器上到底发生了什么并且它是如何将服务检测结果送到 Nagios 中心服务器上的 (下述步骤与前面的参考图中的数字相对应)：

- 在分布服务器完成一次服务检测后，它会执行所定义的强迫型服务处理命令，就是那个 [ocsp_command](#) 变量指向的命令。在本例中，指向的是 **/usr/local/nagios/libexec/eventhandlers/submit_check_result** 脚本。注意在 **submit_check_result** 命令中将四个信息传给了脚本：服务所绑定的主机名、服务描述、服务检测返回结果以及服务检测的输出正文；
- 该 **submit_check_result** 脚本将服务检测信息 (主机名、服务描述、服务检测结果和输出正文) 导入到 **send_nscd** 客户端程序；

- 由 `send_nsca` 程序传送服务检测信息到位于中心服务器上的 `nsca` 守护进程；
- 位于中心服务器上的 `nsca` 守护进程接收到服务检测信息后写入到外部命令文件里让 Nagios 服务来后续处理；
- 位于中心服务器的 Nagios 服务进程读入外部命令文件并处理这些来自于远程分布式服务器上的强制服务检测信息。

9.2.7. 中心服务器配置

已经看过分布式服务器如何配置，下面回到中心服务器来做配置。为保障集中处理运行，中心服务器通常只在一台独立运行 Nagios 机器上做配置工作，设置过程如下：

- 中心服务器安装并配置 Web 接口(可选的，但推荐这么做)；
- 将中心服务器上的 `enable_notifications` 域设置为 1，这将使能通知功能(可选的，但推荐这么做)；
- 将中心服务器的 `自主服务检测` 功能关闭(可选的，但推荐这么做一见下面的提示)
- 使能中心服务器的 `外部命令检查` 项(必须的)；
- 使能中心服务器的 `强制服务检测` (必须的)；

在配置中心服务器时，有另外三件重要的事情要记得做：

- 中心服务器里必须要有**全部分布服务器上的全部服务**定义。如果没有定义服务对象，在中心的 Nagios 将会忽略那些没有正确配置的强制检测服务的检测结果；
- 如果中心服务器只负责处理来自分布服务器的服务检测结果，可以在程序层面关闭自主检测，设置 `execute_service_checks` 域值为 0 即可。如果需要设置中心服务器做些自主检测来监控一些它自己专属的服务(不归分布服务器管)，在分布服务器里这些服务对象的定义里的 `enable_active_checks` 选项值设为 0。这将阻止 Nagios 对这些服务做自主检测。

很重要的是要么关闭程序层面的自主检测设置，要么把分布服务器里的服务对象定义里的 `enable_active_checks` 选项设为 0，这将确保自主服务检测不会在一般状况下被执行。这些服务将会以正常的检测周期间隔(3 分钟或 5 分钟等)来重制订计划表，但不会被真的执行。在 Nagios 运行过程中这种重制订计划的循环会不断地重复，后面我会解释一下为何要这么做。

就这么多，很容易，是吧？

9.2.8. 强制检测中的问题

一般意义上讲，中心服务器将只用强制检测方式来做监控。完全依赖于强制检测来做监控的主要问题是 Nagios 必须依赖于其他东西来提供监控系统数据。如果发送强制检测结果的远程主机宕机或不可达时会怎么样？如果 Nagios 不自主检测主机服务，它怎么会知道有故障发生了？

幸运的是有办法来处理这种类型的故障...

9.2.9. 刷新检测

Nagios支持对服务的检测结果做刷新检测的特性。更多的有关刷新检测信息可以在 [这篇文档](#) 查看。该特性可以在那些远程主机可能停止送出强制服务检测结果的地方提供针对中心服务器提供保护。“刷新检测”的目的就是要确保服务检测要么可由分布式服务器以规格化的方式提供检测数据要么由其中心服务器在必要情况下自主地进行检测。如果分布式服务器所提供检测结果被判定“陈旧”，Nagios将被配置为自中心监控服务器强制地对那个服务发出自主检测。

那么如何来做呢？在中心服务器上需要对那些由分布式服务器所负责监控的服务做如下配置改动...

- 服务对象定义里的 **check_freshness** 选项设为 1，这将打开针对该服务的“刷新检测”特性；
- 服务对象定义里的 **freshness_threshold** 选项须设定为一个以秒为单位的数值，该值反应出由分布式服务器所提供的检测数据将应该以什么样频度来提供出来；
- 在对象定义里的 **check_command** 选项应指向一个有效的命令，当中心监控服务器需要执行自主检测时可以用此命令来执行检测。

Nagios定期地对那些打开了“刷新检测”服务的检测结果进行刷新情况检查。服务对象定义里的 **freshness_threshold** 选项指定了服务检测结果应该在何时间内刷新。例如，如果某个服务里这个选项值是 300，Nagios将会对当前时间 300 秒（即 5 分钟）之前的检测结果判定为“陈旧”。如果没有指定服务对象里的 **freshness_threshold** 值，Nagios将自动地计算出一个刷新闻隔门限，要么按照 **normal_check_interval** 要么按 **retry_check_interval** 来计算，这取决于服务当时所在什么样的 [状态类型](#)。一旦服务检测结果被判定为是“陈旧”，Nagios将使用服务定义里 **check_command** 指定的命令来执行一次服务检测，这当然是自主服务检测。

一定要记得在中心服务器的服务对象定义里指定 **check_command** 选项以便进行从中心服务器发出自主服务检测命令。一般情况下，该检测命令不会被执行（因为自主检测在程序层面被关闭了或者是在服务对象定义里关闭了自主检测）。当对结果的刷新检测功能打开时，Nagios 将会运行自主检测命令 **即便是自主检测被程序层面被关闭或是服务对象里被关闭也会做（自主检测）**。

如果无法在中心监控主机上定义出针对服务的自主检测命令（也或许是超出了责任范围），可以在服务定义的 **check_command** 选项里设一个只是返回紧急状态的脚本来简单地填补该命令。这有个例子... 假定定义了一个名叫 'service-is-stale' 的命令并在服务对象定义里加在了 **check_command** 选项值里，这个定义可能看起来象这样子...

```
define command{  
    command_name    service-is-stale  
    command_line    /usr/local/nagios/libexec/staleservice.sh  
}
```

这个 stale.service.sh 脚本在放在/usr/local/nagios/libexec 目录下，内容可能是这样的：

```
#!/bin/sh

/bin/echo "CRITICAL: Service results are stale!"

exit 2
```

当 Nagios 发现并判定服务检测结果是陈旧的并运行 service-is-stale 命令来做自主检测时，/usr/local/nagios/libexec/stale.service.sh 脚本将执行并返回一个紧急状态，通常情况下将会引发出一个故障通知动作，那么这样就可以知道有故障产生了。

9.2.10. 执行主机检测

此时已经知道了如何从分布服务器获取服务的强制检测结果，这意味着中心服务器不会用自主服务检测机制来取得结果，但怎么来做主机检测呢？仍旧需要做主机检测，但如何做呢？

即然主机检测一般可以折中地只做一小部分自主监控(除非有必要一般是不做的)，我推荐让中心服务器来做点主机自主检测。也就是说，在中心服务器上定义做主机自主检测，这个与分布式服务器上的相同(一般情况下是完全一样的，就象没有做分布式监控一样)。

也可以使用强制主机检测(阅读 [这篇文档](#))，在分布式监控环境下也可以利用这一功能特性，但这可能会导致一些内在问题。最大的问题是Nagios将无法正确地处理强制主机检测结果里的故障状态(宕机与不可达)。这就是说如果监控服务器上的父子节点关系不同(通常情况下因为所处位置不同会有所不同)时，中心服务将会得到一张错误主机当前状态的视图。

如果真的需要从分布式服务器发送强制主机检测结果给中心服务器，确定要做如下配置：

- 中心服务器里打开 [强制主机检测](#)开关(必须的)；
- 分布服务器打开 [强迫型主机](#)处理开关；
- 分布服务器里定义好了 [强迫型主机处理命令\(ochp\)](#)。

这个强迫型主机处理命令(ochp)被用于处理主机检测结果，与强迫型服务处理命令(ocsp)的执行方式一样，这个方式在前面已经说明过了。为了保存强制主机检测结果即时更新，还要打开主机的 [刷新检测](#)的功能开关(这个与前面所讲的服务刷新检测相似)。

9.3. 冗余式与失效式网络监控

9.3.1. 介绍

本文档介绍几个在几种类型的网络层上实现冗余主机监控的场景。利用冗余主机，可在运行 Nagios 的核心主机宕机或是一部分网络变为不可达等等情况发生时维持对网络的监控。

注意

如果只是学习如何使用Nagios，建议你先不要去尝试冗余方式运行，直到你真正熟悉所列出的冗余方式运

行所应有的 [先决条件](#)。冗余式运行有些难以理解而且比较难以正确地配置实现。

9.3.2. 索引

- [先决条件](#)
- [样例脚本](#)
- [场景 1—冗余监控](#)
- [场景 2—失效监控](#)

先决条件

在用 Nagios 实现冗余监控之前，需要熟悉以下内容...

- 主机与服务 [事件处理](#)的实现；
- 用脚本来解决Nagios的 [外部命令](#)问题；
- 在远程主机上执行插件，要么用 [NRPE外部构件](#)要么是其他方式；
- 用 `check_nagios` 插件来检测 Nagios 进程的状态。

样例脚本

本文档里所用脚本可以在 Nagios 发行包的 `eventhandlers/`子目录里找到。可以修改这些脚本以适合你的监控系统...

场景 1—冗余监控

介绍

在网络中这是一个很容易实现的冗余监控主机的方法，也很初级，它只是对有限数量的故障起到防止作用。为更巧妙地实现冗余监控需要更复杂些的安装配置，更有效的冗余监控是利用不同的网段等方式来实现冗余。

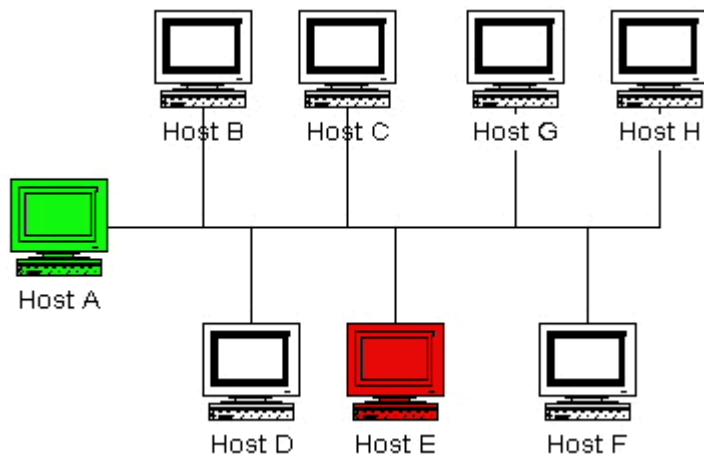
目标

这种冗余形式只是简单实现冗余。把监控“主”机和“从”机针对网络里相同的主机与服务进行监控。一般情况下只是由“主”机把有关故障信息以通知形式送给联系人。希望由“从”机运行 Nagios 并在如下情况发生时完成把故障通知给联系人的工作：

- 运行 Nagios 的监控“主”机宕机
- 因某种原因“主”机上的 Nagios 进程停止运行

网络层示意图

下图给出非常简单地网络安装实现的方式。在该场景，假定主机 A 和主机 E 都运行 Nagios 并且都监控全部主机。机器 A 认定是监控“主”机而机器 E 是“从”机。



初始化程序设置

“从”机(机器E)初始化它的 [enable_notifications](#)域是非使能的,这样阻止了它因任何主机与服务问题而送出通知。还要保证“从”机把 [check_external_commands](#)域使能,这就很容易地实现了...

初始化配置

下面需要考虑在“主”机和“从”机上的 [对象配置文件](#)差异...

假定已经在“主”机(机器 A)上设置好针对图中全部的主机和服务进行监控。“从”机(机器 E)也需要做相同的设置,还需要做些额外的配置修改...

- 针对机器A的对象定义(在机器E的配置文件里)需要给定一个主机的 [事件处理](#)定义,把该事件处理称为handle-master-host-event;
- 机器 E 的配置文件需要定义一个服务来监控机器 A 上 Nagios 服务进程的状态。假定是由机器 A 上的 [check_nagios](#) 插件来完成这个服务检测工作。可以查阅这个 FAQ 里的描述方法来实现它(需要更新!);
- 在机器A上的配置里针对Nagios进程的服务定义时需要有一个 [事件处理](#)定义,把该事件处理称为handle-master-proc-event;

注意一点,机器 A(“主”机)对机器 E(“从”机)一无所知,在该场景里只是没这个必要。当然也可以让机器 A 来监控机器 E,但这这与冗余监控一点关系都没有...

事件处理命令的定义

需要停顿一下来说明在“从”机上的事件处理命令应该是什么样子的,这有个例子...

```
define command{
    command_name    handle-master-host-event
    command_line    /usr/local/nagios/libexec/eventhandlers/handle-master-host-event
$HOSTSTATE$ $HOSTSTATETYPE$
}
```

```

define command{
    command_name      handle-master-proc-event
    command_line      /usr/local/nagios/libexec/eventhandlers/handle-master-proc-event
$SERVICESTATE$ $SERVICESTATETYPE$
}

```

假定已经把事件处理脚本放在了 `/usr/local/nagios/libexec/eventhandlers` 目录里，也可以把脚本放在其他你想放的位置上，但需要对这些例子做些修改才能用。

事件处理的脚本

OK，看一下事件处理脚本的内容...

主机事件处理脚本 (handle-master-host-event):

```

#!/bin/sh

# Only take action on hard host states...

case "$2" in
HARD)
    case "$1" in
DOWN)

        # The master host has gone down!

        # We should now become the master host and take
        # over the responsibilities of monitoring the
        # network, so enable notifications...

        /usr/local/nagios/libexec/eventhandlers/enable_notifications

        ;;

UP)

        # The master host has recovered!

        # We should go back to being the slave host and
        # let the master host do the monitoring, so
        # disable notifications...

        /usr/local/nagios/libexec/eventhandlers/disable_notifications

        ;;

esac

```

```
;;  
  
esac  
  
exit 0
```

服务事件处理脚本(handle-master-proc-event):

```
#!/bin/sh  
  
# Only take action on hard service states...  
  
case "$2" in  
HARD)  
    case "$1" in  
CRITICAL)  
        # The master Nagios process is not running!  
  
        # We should now become the master host and  
  
        # take over the responsibility of monitoring  
  
        # the network, so enable notifications...  
  
        /usr/local/nagios/libexec/eventhandlers/enable_notifications  
  
        ;;  
  
WARNING)  
  
UNKNOWN)  
  
        # The master Nagios process may or may not  
  
        # be running.. We won't do anything here, but  
  
        # to be on the safe side you may decide you  
  
        # want the slave host to become the master in  
  
        # these situations...  
  
        ;;  
  
OK)  
  
        # The master Nagios process running again!  
  
        # We should go back to being the slave host,  
  
        # so disable notifications...  
  
        /usr/local/nagios/libexec/eventhandlers/disable_notifications  
  
        ;;  
    esac  
done
```

```
    esac

    ;;

esac

exit 0
```

这样会有什么效果？

“从”机(机器 E)初始不发通知，因而在“主”机(机器 A)上的 Nagios 进程运行时不会送出任何有关主机与服务的通知。

在“从”机上的 Nagios 进程将在下列情况时会变为“主”监控状态...

- “主”机(机器 A)宕机并且 **handle-master-host-event** 主机事件处理命令脚本被执行；
- “主”机(机器 A)上的 Nagios 进程停止运行并且 **handle-master-proc-event** 服务事件处理命令脚本被执行；

当“从”机(机器 E)上的 Nagios 进程打开了通知开关时，它将送出任何一个有关主机与服务的故障与恢复的通知，在这一点上，机器 E 扮演着把主机与服务故障通知给联系人的后备冗余的角色！

机器 E 上的 Nagios 进程会重新回到“从”机状态，当如下情况发生时...

- 机器 A 恢复并且 **handle-master-host-event** 主机事件处理脚本命令被执行；
- 机器 A 上的 Nagios 进程服务恢复并且 **handle-master-proc-event** 服务事件处理脚本命令被执行；

当机器 E 上的 Nagios 进程的通知功能关闭时，它将不会送出任何有关主机与服务故障与恢复的通知。在这一点上，机器 E 扮演着机器 A 的把主机与服务故障通知到联系人的后备冗余的角色。当系统起动后，所有这些已经各就各位了！

切换时间差

Nagios 的冗余方式并不完善，一个显而易见的问题是在“主”机宕机和“从”机接管之间存在切换时间差。它受如下因素影响：

- 从“主”机宕机到“从”机首次发现一个故障之间的时间；
- “主”机真正地验证了一个故障存在(在从机上通过多次服务与主机检测的重试验证)所需时间；
- 事件处理所需要的执行时间和下一次 Nagios 检查外部命令文件的时间；

可以减小切换时间差，通过修改这些内容...

- 确保在机器 E 上以较高频度执行对 Nagios 服务的一个或多个检测。可以修改服务定义里的 **check_interval** 和 **retry_interval** 选项来实现；
- 确保在机器 E 上快速地完成针对机器 A 的重试检测。可以修改主机定义里的 **max_check_attempts** 选项来实现；

- 在机器E上增加 [外部命令](#) 检测的处理频度。可以修改在主配置文件里的 [command_check_interval](#) 选项来实现；

当机器 A 上的 Nagios 服务恢复时，同样会有一个切换时间差，完成机器 E 回到“从”机的运行状态，它受如下因素影响：

- 从机器 A 恢复到机器 E 上的 Nagios 服务检测到服务已经恢复所需时间；
- 在机器 A 上执行事件处理后的一时刻起到机器 E 上的 Nagios 服务最近一次处理外部命令文件之间的时间；

监控的冗余监控运行状态的真正切换时间差取决于定义有多少服务、服务以什么间隔被检测和一系列偶然因素。无论如何，这总比没有要好。

特例

有件事需要注意... 如果机器 A 宕机了，机器 E 将使能通知功能并且接管了故障通知。当机器 A 恢复了，机器 E 将关闭通知功能。如果当机器 A 恢复了但是机器 A 上的 Nagios 服务并没有正确地启动起来话，将会有段时间，在这段时间内不会有任何主机故障的通知被送给联系人！幸运的是，Nagios 的服务检测逻辑会处理这种情况，在机器 E 上的 Nagios 在最近一次的处理机器 A 的状态时，它将发现机器 A 的 Nagios 服务没有运行。机器 E 将会再次打开通知功能并接管对联系人的故障通知的工作。

统计哪一个主机完成针对网络监控的真正时间是很困难的，很明显，可以通过增高针对机器 A 上的 Nagios 服务的检测频度(在机器 E 上配置)来最小化这一时间间隔，另外就完全是偶然因素的，但整体上的“阻塞”时间并不太多。

场景 2—失效性监控

介绍

失效性监控与冗余监控相似，但有很少一点不同(冗余监控在 [场景 1](#) 讨论)。

目标

失效监控的最基本目标是“从”机上的 Nagios 机器会在“主”机上的 Nagios 运行时一直保持空转。如果“主”机上的 Nagios 进程服务停止(或者机器宕机)时，“从”机上的 Nagios 将会开始对全部网络监控的工作。

虽然在 While the method described in [场景 1](#) 里所描述的方法将会在“主”机宕机时不断地收到通知，但会有些毛病。最大的问题是“从”机也会在“主”机在相同的时间里对网络中的全部主机和服务执行检测！如果在被监控主机上定义有很多服务时这会给主机产生额外的流量及负荷。这里有绕开这个问题的办法...

初始化程序设置

在“从”机上关闭自主检测和通知功能，通过修改 [execute_service_checks](#) 和 [enable_notifications](#) 选项来实现。这将使得在“主”机运行主Nagios服务正常的情况下阻止“从”机对主机与服务的检测和送出通知。同时要确保在“从”机上打开了 [check_external_commands](#) 选项。

主服务进程检测

在“从”机上设置一个定时作业(cron job)来周期性地(比如每分钟)运行一个脚本，该脚本用于对“主”机的Nagios服务进行检测(在“从”机上使用[check_nrpe](#)插件，同时，在“主”机上运行 [nrpe守护服务](#)和 [check_nagios](#)插件)，该脚本将检查由[check_nrpe](#)插件所返回的结果码。如果返回结果码是个非正常状态，脚本将写入一个切换冗余命令到 [外部命令文件](#)中以使能通知功能和自主检测。如果插件返回一个正常状态，脚本将往外部命令文件里写一个命令以关闭通知功能和自主检测功能。

通过这么做，将会使得在某个时间内只会有一个 Nagios 监控进程在真正运行，这比用两次监控网络的方式效率更高。

还要注意的，没有必要象是在 [场景 1](#) 里那样定义主机与服务的事件处理，因为处理方式完全不同。

其他问题

此时已经完成了一个基本的失效监控系统的安装设置，然而，要使它工作更顺畅还有几件事要考虑。

这么设置后的最大问题是，“从”机在其开始监控工作时它并没有当前任何主机与服务的状态信息。解决这一问题的一个方法是在“主”机上使能 [强迫型服务处理命令](#) 并且把全部的服务检测结果用 [nsca外部构件](#) 送到“从”机上。“从”机就会在承担监控工作时已存有所有服务的最新状态信息。因为“从”机上没有打开自主服务检测，它将不会自主地运行任何服务检测。然而，它将在必要时执行主机检测。也就是说，“主”和“从”都会在必要时执行主机检测，这不是大问题，因为监控的主要工作大部分是服务的检测。

9.4. 大型安装模式的变化

9.4.1. 介绍

用户在使用Nagios大型安装模式将会有许多好处，使用 [use_large_installation_tweaks](#) 配置选项。使能这个选项将使Nagios守护程序将进行某些短路以使系统负载更低且性能最好。

9.4.2. 影响

当你在主配置文件中使能了 [use_large_installation_tweaks](#) 配置选项，将会使Nagios守护进行做如下变化：

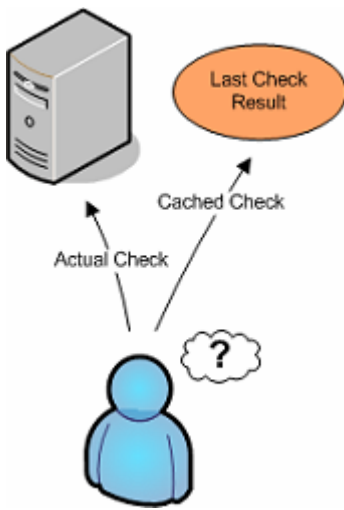
- 在环境中不能使用汇总类的宏—[汇总宏](#)在环境中将不能使用。这些宏的计算在大型安装时会非常地集中消耗时间，因此它们在此时是不能使用的。但如果你在脚本中传递这些参数时，这些汇总性的宏仍旧可用于规格化的宏而加入脚本。
- 内存清理有所不同—通常 Nagios 在子进程退出时会释放子进程分配的内存，这是个好习惯，然后在许多安装模式下并不需要，因为许多操作系统将会很小心地处理进程退出时的内存。操作系统

倾向于自主地释放内存而不是由 Nagios 来做，这样更快，因而 Nagios 不再试图释放子进程的内存空间，如果你使能了这个配置选项的话。

- 派生 fork() 检查更少—通过 Nagios 会在主机和服务检测时做两次派生。这样做是因为 (1) 确保受阻的插件有一个较高的进程等级捕获错误信号或进入异常；(2) 让操作系统来对那些退出子进程的下级进程做清除处理。额外的派生并不是真有必要，所以在你使能这个配置选项后它会跳过额外派生动作。Nagios 将自行清理子进程的退出（而不是等到操作系统来做它）这使得 Nagios 在这种安装模式下显著地降低负载。

9.5. 缓存检测

9.5.1. 介绍



应用了缓存检测机制可以显著地改善 Nagios 监控逻辑的性能。缓存检测的作用是，当 Nagios 发现可以利用最近一次检查结果来替代这次检测时，Nagios 会放弃执行一次主机与服务的检测。

9.5.2. 只为按需检测使用

应用缓存检测机制对于通常的规格化编制的主机与服务检测的性能不会有明显改善。缓存检测只是对于主机与服务的按需检测的性能有显著改善。预定的计划性检测可以确保主机与服务的状态更新规范化，它使得在不久的将来，它的检查结果最有可能被缓存检测所利用。

作为参考，要做主机的按需检测...

- 当绑定于主机上的服务状态发生了变更；
- 部分检查内容由于需要做 [主机可达性](#) 逻辑判断；
- 部分是由于需要做 [主机依赖性检测的前处理](#)。

要做服务的按需检测...

- 是由于需要做 [服务依赖性检测的前处理](#)。

注意

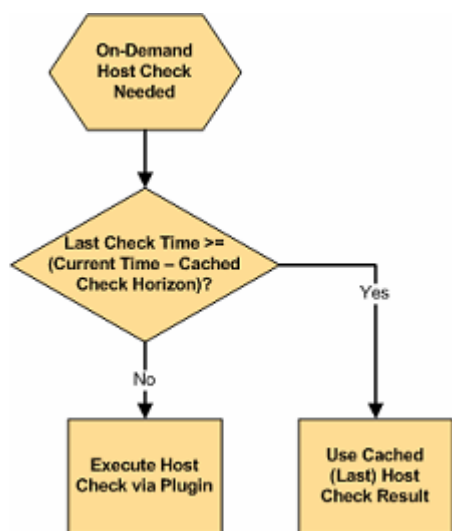


除非你打开了服务依赖性检查，Nagios 将不会使用缓存检测机制来改善服务检测的性能。不必担是，这只是通常情况下是这样。缓存主机检测并不是个极大提升性能做法，每个人都应看到它只是有益于提高性能。

9.5.3. 缓存检测是如何工作的？

当 Nagios 需要做一个主机与服务的按需检测时，它将做一个判定，是否要利用缓存检测结果还是要真的去用插件来做一次检查。这取决于这次主机与服务的最近一次检测结果是否发生于最近的 X 分钟之内，这里 X 是缓存主机与服务结果的时间长度。

如果最近一次检测的时间刚好在指定缓存检测结果的时间内，Nagios 将会利用最近一次针对该主机与服务检测结果而不会真的去做一次检测。如果该主机与服务的检测没有做过，或是最近一次检测结果的时间超出缓存检测的时间深度，Nagios 将会用插件对该主机与服务来做一次新的真正的检查。



9.5.4. 这将到底意味着什么？

Nagios 做按需检测是由于它认为有必要及时地知道该主机与服务在**那一时间**里的状态。利用缓存检测结果将使得 Nagios 可以认为最近一次检测结果是“足够好用”的当前主机与服务的状态，并且认定真的没有必要再去做一次该主机与服务重检测。

缓存检测的时间深度告诉 Nagios 在多大的时间内检测的结果是值得信赖地反应出了当前的主机或服务状态。比如，时间深度设置是 30 秒，那么在最近的 30 秒之内的主机与服务的检测结果就可以被认为是当前的主机与服务状态结果。

Nagios 的可用缓存结果数量与需要执行按需检测次数之比被认为是缓存检测的“击中率”。增加缓存检测的时间深度直到该值等于规格化检测的时间间隔，在理论上可以实现缓存检测的击中率到 100%。在这种情况下，全部的按需检测都可以从缓存检测的结果中提取，多高的性能改善啊！但是真的么？可能并非如此！

缓存检测结果信息的可信度随时间而降低。高的缓冲击中率需要加长认定为“合法”结果的缓存时间。但各种网络场景变换很快，而且没有任何可以担保在 30 秒之前处于正常状态的服务当前也是处于正常的。因而不得不取个折中—信任度与速度之间取折中。如果要提高缓存结果的时间深度，就不得不要冒着缓存结果应用于监控逻辑之中信任度降低的风险。

Nagios 将最终判定出全部主机与服务的正确的状态，因此即便在缓存中的检测结果相对于其真实情况有可能是不可信的，Nagios 也只是会在一个短时间内在不正确信息之下工作。在这么短时间内的不可信状态信息对于管理员是个讨厌的事情，因为管理员可能会收到故障通知但它不久就不再有了。

对于 Nagios 用户而言，没有一个标准来检验缓存检测的时间深度或缓存击中率是可接受的。有些需要一个短暂的检测缓存时间深度设置和一个相对低的缓存击中率，而另一些则想要更长些的缓存时间和较高缓存击中率(当然会相对低的状态可信度)，更有甚者希望完全不用缓存检测而只要 100%可信度。测试不同的缓存检测时间窗口大小以及对状态信息可信度的影响将只是少数人想做的，他们只想得到在其自身环境下的“正确”取值。更多的信息见下面讨论。

9.5.5. 配置变量参数

如下的变量将决定用于缓存主机与服务检测结果的时间窗口值，在哪个范围内的检测结果可用于主机与服务的检测结果：

- [cached_host_check_horizon](#)变量控制缓存主机检测结果；
- [cached_service_check_horizon](#)变量控制缓存服务检测结果；

9.5.6. 优化缓存效率

为了应用缓存检测机制达到最高效率，应该做如下工作：

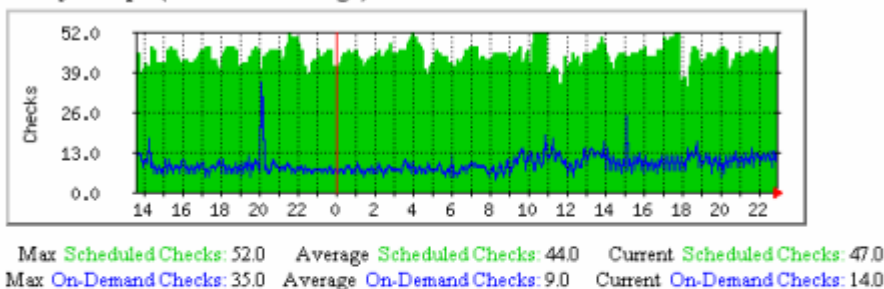
- 编制规格化的主机检测计划；
- 使用 MRTG 来绘制统计状态图，做出 (1) 按需检测图 (2) 缓存检测图；
- 调整缓存检测的时间深度以适合当前情况。

在编制主机规格化检测计划时，可以把 [主机对象定义](#)里的`check_interval`域指定一个大于 0 的值，如果这样做，还应保证将`max_check_attempts`域设置得大于 1，否则会引起一个性能突降，这个性能突降在 [这篇](#)文档里有说明。

给缓存检测的时间深度取值的一个较好方式是把有多少Nagios的按需检测被执行和有多少是取自于缓存检测结果这两个值做比较。[nagiosstats](#)工具将提供缓存检测的相关信息，这些信息可以 [用MRTG绘制图表](#)。样例的MRTG图表见下面，图中给出了缓存中取结果次数与实际执行检测的次数。

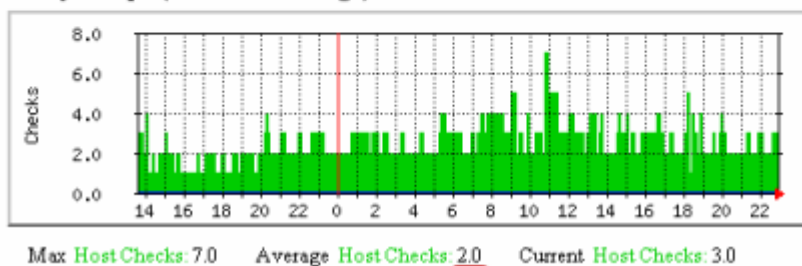
Active Host Checks

'Daily' Graph (5 Minute Average)



Cached Host Checks

'Daily' Graph (5 Minute Average)



上述监控安装运行而产生图示的事先设置有：

- 共计有 44 台主机，它们全部用计划检测间隔来检测；
- 平均(规格化时间表内的)主机检测间隔是 5 分钟；
- [cached_host_check_horizon](#)值是 15 秒

第一张 MRTG 图表显示了有多少规格化计划主机检测与实际做了多少缓存主机检测的比较。在这例子中，每 5 分钟平均会有 53 次主机检测，其中有 9 次是按需主机检测(占到检测总数的 17%)；

第二张 MRTG 图表显示了沿时间轴上会有多少缓存主机检测结果产生。在这例子中，每 5 分钟平均会有 2 次缓存主机检测；

记住，缓存检测只是对按需检测起作用。基于图中的每 5 分钟的平均值，可见 Nagios 是每 9 次应做的按需检测中有 2 次是使用缓存检测结果。这看起来不多，但图中只是给出了一个小型的监控环境的结果，考虑到 2 比 9 就是 22%的性能提高的话，就会明白将会在一个大型监控环境下将会显著地改善性能如果把主机检测的时间深度加大的话会提高缓存结果的击中率，但也会同时降低了缓存主机状态信息的可信度。

一旦有了几小时乃至几天的 MRTG 图表，就可以看出主机与服务的检测中有多少是插件执行而有多少是利用的缓存结果。利用这些图表信息来调整缓存检测的时间深度以适合当前环境，不断地利用 MRTG 图表来监视缓存检测时间深度变量对缓存检测统计在时间维度上的影响情况，并在需要的时候清掉重新来过。

9.6. 状态追踪

9.6.1. 介绍

状态“追踪”是个并不通用的功能特性。使能了它，即便是主机与服务的状态没有变化的情况下也可对状态检测的结果产生日志记录。当对部分主机与服务开启了状态追踪功能时，Nagios 将会仔细地监控这些主机与服务并记录下任何有关的检测结果的输出内容。可见，这对于日后分析日志文件很有帮助。

9.6.2. 它是如何工作的？

在通常情况下，只有在主机与服务的状态与最近一次的检测结果发生变化时才会对检测结果产生日志记录。只有很少情况例外，这是个规则。

如果对部分的主机与服务使能一种或多种的状态追踪功能，Nagios 将在本次检测结果与前一次检测有差异的情况下产生输出结果的日志记录。下面例子中有八次连续的服务检测，看看有什么差别：

表 9.1.

服务检测号	服务状态	检测的结果输出	通常日志	有跟踪的日志
x	正常	RAID array optimal	—	—
x+1	正常	RAID array optimal	—	—
x+2	告警	RAID array degraded (1 drive bad, 1 hot spare rebuilding)	✓	✓
x+3	紧急	RAID array degraded (2 drives bad, 1 host spare online, 1 hot spare rebuilding)	✓	✓
x+4	紧急	RAID array degraded (3 drives bad, 2 hot spares online)	—	✓
x+5	紧急	RAID array failed	—	✓
x+6	紧急	RAID array failed	—	—
x+7	紧急	RAID array failed	—	—

在上述的一系列检测中，一般的日志方式只可以看到对此次网络事故的两条记录。第一条是在 x+2 次时服务从正常变换为告警状态，第二条是在 x+3 时，状态由告警转变为紧急。

无论何原因，你可能需要在日志文件里对此次网络事故保存有完整的历史记录。可能是有助于你向主管解释事故情况失控是何等的快，也可能是在酒巴里与一堆人谈笑有个好话题...

那么，如果对该服务的紧急状态使能了追踪功能，将会在 x+4 和 x+5 时多两个日志事件。这是为什么呢？当打开了状态追踪功能，Nagios 将对每一次检测结果进行检查，看此次结果与最近一次结果是否存在

差异。如果输出状态不同而且服务的状态并没有改变，那么这次新检测结果的输出就会被日志文件记录下来。

状态追踪最直接的例子是对一个 WWW 服务的监控检测。如果 check_http 检测插件第一次检测时，因一个 404 错误返回一个告警状态，之后因为期望匹配串没有找到返回一个告警，这可能是你需要知道的情况。如果没有使能 WWW 服务的告警状态追踪，只会得到第一次转入告警状态时日志 (404 错误而产生的)，但自此之后是因为在返回页面里没有匹配到指定字符串而产生的告警状态不是 404 错误，这会让你在打开压缩的日志文件包时不知如何下手。

9.6.3. 需要使能状态追踪么？

首先，必须断定你真的需要在打包的日志文件里寻找故障根源。你可能是只需要对几个主机与服务打开这个功能而非全部。你可能是只需要对某个主机或某个服务的某几个状态进行状态追踪而非全部。比如，你需要对一个服务的告警和紧急状态进行状态追踪但不需要对正常和未知进行追踪。

对部分主机或服务的状态追踪是否打开使能也跟对该主机与服务做检测的插件有关。如果对于某种状态，检测插件总是返回相同的输出字符串，那就没理由为该服务打开状态追踪功能（——打开也没用，全是一个内容）。

9.6.4. 如何使能追踪？

可以在 [主机与服务对象定义](#) 的 `stalking_options` 域设置状态追踪功能的打开或关闭。

9.6.5. 状态追踪与可变服务有何不同？

状态追踪与 [可变服务](#) 相似。但可变服务会引发通知动作和事件处理的执行，而状态追踪纯粹是为了产生详细日志。

9.6.6. 限制与告诫

需要当心在使用状态追踪时会有一些潜在问题。主要会发生在几个 [CGI 模块](#) (历史状态、报警汇总等) 的报告功能中。因为状态追踪会增加一些额外的日志，这些会增大产生故障日志的次数，而生成报告时是以这些数据为基础的。

作为一条通行规则，建议在没有考虑完善之前不要打开主机与服务的状态追踪功能，除非，你确实想要用这个功能。

9.7. 集群主机和集群服务的监控

9.7.1. 介绍

有些人咨询有关集群主机和集群服务的监控，因而写本文档来解释如何来做，希望这些简单明瞭。

首先来解释一下“集群”。用例子来说明较容易理解。有 5 台主机来提供 DNS 解析服务，如果一台宕机，这不算什么，因为仍旧有几台机器在提供 DNS 服务。如果需要监控这个 DNS 服务，那么就有 5 个 DNS 服务器，这种情况下，这个 DNS 服务被认为是一个 **集群服务**，它有 5 个独立的 DNS 服务构成。虽然需要各

自独立地对服务进行监控，但更关心的是整体的 DNS 服务集群能否正常工作，而不是某个独立的服务工作情况。

如果你有一个主机群来提供高可靠性服务(集群)解决方案，这种情况被认为是一个**集群主机**。如果一台主机宕机，另一个将接管全部失效主机的工作。一个提示，如果想配置一个Linux集群系统，可以访问 [高可靠性Linux集群项目](#)。

9.7.2. Plan of Attack

有几步来监控集群服务和集群主机。下面将尽可能简单地说明。监控集群主机和集群服务有两步要做：

- 监控集群内的元件
- 监控一个集合体

监控集群中的主机与服务元件比较容易。事实上可能你已经做过了。对于集群服务，只要确保已经实现了对集群服务中的每个服务都已经处于监控状态，如果是一个 5 台 DNS 服务组成的集群，要保证定义出了 5 台域名解析服务对象(可能会用到 `check_dns` 插件)。对于集群主机，要确保对集群主机中的每一台都有对象定义(同样必须要给每个主机至少要绑定一个服务)。

重要

可能要关闭集群主机或集群服务里每个元件的报警通知功能。虽然每个元件没有独立送出通知，但仍旧可以在 [当前状态CGI](#) 模块里独立地查看每个集群元件状态显示。这样可能有助于查找集群服务的故障根源。

监控集群整体可以使用对集群每个元件检测的缓存检测结果。虽然也可以在集群检测的时候对全部元件进行再检测，但有缓存结果的情况下为什么要再次浪费带宽和资源来再做一遍呢？缓存结果在哪里？在 [状态文件](#) 中保存了集群中的每个元件的缓存检测结果(假定已经对每个元件进行监控)。那个

`check_cluster` 插件被设计为在状态文件中取出缓存检测结果来完成对集群检测。

重要

虽然没有对集群每个元件使能通知功能，但仍旧可以完成对集群整体的检测。

9.7.3. 使用集群检测 `check_cluster` 插件

那个 `check_cluster` 插件被设计为从集群中每个独立的集群元件状态结果中提供状态信息来生成集群整体状态。

`check_cluster` 插件可以在Nagios插件软件包 (<http://sourceforge.net/projects/nagiosplug/>) 的发行目录中找到。

9.7.4. 监控服务集群

假定要对一个由 3 台 DNS 服务组成的了冗余域名解析服务群进行监控。首先，在做为一个集群监控之前，已经可以各自独立地完成对三个域名解析服务的监控，假定是 3 台 DNS 主机分别叫 `host1`、`host2` 和 `host3`，它们上面绑定有名为“DNS Service”的服务。

为了完成对集群监控，需要创建一个“集群”服务对象。然而，在此之前，需要先定义一个对集群服务检测命令配置。假定这个叫做 `check_service_cluster` 命令是这样定义的：

```
define command{
    command_name      check_service_cluster
    command_line       /usr/local/nagios/libexec/check_cluster --service -l
$ARG1$ -w $ARG2$ -c $ARG3$ -d $ARG4$
}
```

现在就可以创建一个叫“cluster”的集群服务对象了，里面用刚刚定义的 `check_service_cluster` 命令来做检测。下面例子将示意如何做。例子中给出了如果有 2 个或 2 个以上的服务处于非正常状态时将会产生一个紧急状态，如果只有 1 个服务处于非正常状态时将产生告警状态，如果全部正常将返回一个集群服务处于“正常”的状态。

```
define service{
    ...
    check_command      check_service_cluster!"DNS
Cluster"!1!2!$SERVICESTATEID:host1:DNS Service$, $SERVICESTATEID:host2:DNS
Service$, $SERVICESTATEID:host3:DNS Service$
    ...
}
```

很重要的一点是，要注意在集群检测命令里的第 4 个命令参数宏 `$ARG4$` 里使用了一个逗号分隔的**按需**服务状态 [宏](#)列表，这个非常重要！Nagios 将当前集群中的每个服务状态 ID 来填充这些按需宏的状态值（用数值而不是字符串），有关按需主机与服务宏的信息可以查阅 [这篇](#)文档。

9.7.5. 主机集群的监控

对集群主机的监控与集群服务相类似，当然最主要的不同的组成集群的是主机而非服务。为完成对集群主机的监控，必须定义一个使用 `check_cluster` 插件的服务对象。该服务将**不属于**集群中的任何一个主机，因为绑定在某个主机上时，当它宕机时会使得集群主机的通知无法送出。把这个集群服务绑定在 Nagios 运行主机上是个好主意。毕竟如果 Nagios 运行可以对集群检查，如果宕掉，也就无法进行监控了（除非你设置了 [冗余监控主机](#)）...

不管如何，假定有一个命令 `check_host_cluster` 的定义是这样的：

```
define command{
    command_name      check_host_cluster
```

```
command_line    /usr/local/nagios/libexec/check_cluster --host -l $ARG1$  
-w $ARG2$ -c $ARG3$ -d $ARG4$  
}
```

假设是个 3 台主机(分别命名为“host1”、“host2”和“host3”)组成的集群主机。如果想让 Nagios 在发现有 1 台没有运行时送出告警报警, 在 2 台或 2 台以上时送出紧急报警, 那么集群主机的监控会这样来定义:

```
define service{  
    ...  
    check_command    check_host_cluster!"Super Host  
Cluster"!1!2!$HOSTSTATEID:host1$, $HOSTSTATEID:host2$, $HOSTSTATEID:host3$  
    ...  
}
```

很重要的一点是, 要注意在集群检测命令里的第 4 个参数\$ARG4\$是传递了一个逗号分隔的**按需**主机状态 [宏](#)列表。这个很重要! Nagios 将会用集群里当前各个主机状态的 ID 值(数值而不是字符串)来填充按需状态宏。

就这么多。Nagios 将定期地检测集群主机的状态并在状态变换时(假定已经使能了该服务的通知功能)送给你通知。注意对于集群里的每个主机成员, 可能需要关闭主机宕机等的通知功能。记住, 对于一个集群来说, 你真正关心的一是某台主机的状态而是集群整体状态。你或许想在主机对象定义里去掉有关不可达通知的功能, 这取决于你的网络层情况以及如何现构建集群。

9.8. 适应性监控

9.8.1. 介绍

Nagios 允许你在运行时对主机和服务进行特定检查时变更命令。我把这种特性称为“适应性监控”。请注意 Nagios 的适应性监控对于 99% 的用户是不需要的, 但可以让你做些有趣的事情。

9.8.2. 什么可以改?

在运行时, 如下服务检测属性是可以修改的:

- 检测命令和命令行参数
- 检测周期
- 最大检测尝试次数
- 检测周期
- 事件处理命令及命令参数

在运行时，如下主机检测属性是可以修改的：

- 检测命令和命令行参数
- 检测周期
- 最大检测尝试次数
- 检测周期
- 事件处理命令及命令参数

在运行时，如下的全局属性可以修改：

- 全局的主机事件处理命令及命令参数
- 全局的服务事件处理命令及命令参数

9.8.3. 适应性监控的外部命令

为了在运行时改变全局的、主机的或服务的属性，你需要给出恰当的 [外部命令](#) 给 Nagios，在 [外部命令文件](#) 中设置。表格列出的不同属性可以完成对各自不同的属性进行修改。

一个给适应性检测而制作的完整外部命令列表（包括如何使用样例）可以在如下 URL 中找到：

<http://www.nagios.org/developerinfo/externalcommands/>

注意



注意以下内容：

- 当修改检测命令、检测周期或事件处理句柄时，很重要的一点是在 Nagios 启动之前要注意给这些新值定义好。如果在 Nagios 启动之后，任何试图修改这些设置的尝试都会被忽略。
- 你可以给指定特定的命令参数给这些命令名—只是使用分隔符(!)分开几个命令参数。更多的有关如何定义命令参数的信息可以在 [宏](#) 及应用的文档中找到。

9.9. 强制式主机状态迁移

9.9.1. 介绍

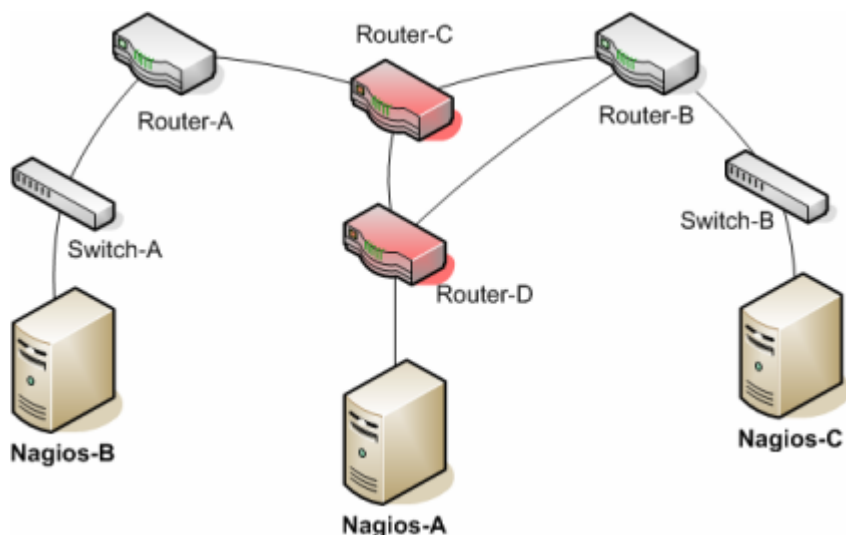
当 Nagios 用强制检测方式从远程源接收主机检测结果时（如其他的 Nagios 分布式实例或分散式安装），由远程资源上报告的主机的状态可能并不能正确地显示在 Nagios 的视图上。在处于分布式或分散式安装方式下由多个 Nagios 实例结果中保证正确地显示主机状态是非常重要的。

9.9.2. 不同的全局视图

下图给出分散式安装的简单例子。图中

- Nagios-A 是主监控服务器并可以对全局的路由器和交换机进行监控。
- Nagios-B 和 Nagios-C 是后备的监控服务器，可以从 Nagios-A 接收强制检测结果。

- 当 Router-C 和 Router-D 处于故障并离线状态。



那么 Router-C 和 Router-D 当前应处于什么状态？结果取决于你访问哪个 Nagios 实例。

- Nagios-A 报告 Router-D 处于宕机且 Router-C 处于不可达
- Nagios-B 报告 Router-C 处于宕机且 Router-D 处于不可达
- Nagios-C 报告全部的路由器处于宕机

每个 Nagios 实例都有不同的网络状态视图，由于后备的监控服务不可以盲目地从主监控主服务器接收主机状态否则它们会得不到正确的网络状态信息。

由于没有转换主监控服务器 (Nagios-A) 的强制主机检测结果, Nagios-C 将认为 Router-D 处于不可达, 除非它自己得到其真的宕机。相同地, 宕机或不可达状态 (从 Nagios-A) 看过去的 Router-C 和 Router-D 的视图会使得 Nagios-B 的视图翻转。

注意



有时你不想让 Nagios 因为远程的源给出的状态而使得视图中显示宕机或不可达状态而翻转你处于“正确”状态的视图，如分布式环境下，你想让中心监控服务器得到不同的分布节点下的不同网络部分的视图。

9.9.3. 使能状态迁移

默认情况下，Nagios 将**不会**自动地用强制检测的宕机和不可达状态来迁移状态。如果你需要必须使能它。

自动地将强制检测结果进行状态迁移受 [translate_passive_host_checks](#) 变量的控制。使能它将使本地的 Nagios 实例接收来自远程资源的宕机和不可达状态迁移而改变显示状态。

第 10 章 Nagios 自身的安全性与性能调优

10.1. 自身安全相关事项

10.1.1. 介绍



This is intended to be a brief overview of some things you should keep in mind when installing Nagios, so as set it up in a secure manner.

Your monitoring box should be viewed as a backdoor into your other systems. In many cases, the Nagios server might be allowed access through firewalls in order to monitor remote servers. In most all cases, it is allowed to query those remote servers for various information. Monitoring servers are always given a certain level of trust in order to query remote systems. This presents a potential attacker with an attractive backdoor to your systems. An attacker might have an easier time getting into your other systems if they compromise the monitoring server first. This is particularly true if you are making use of shared SSH keys in order to monitor remote systems.

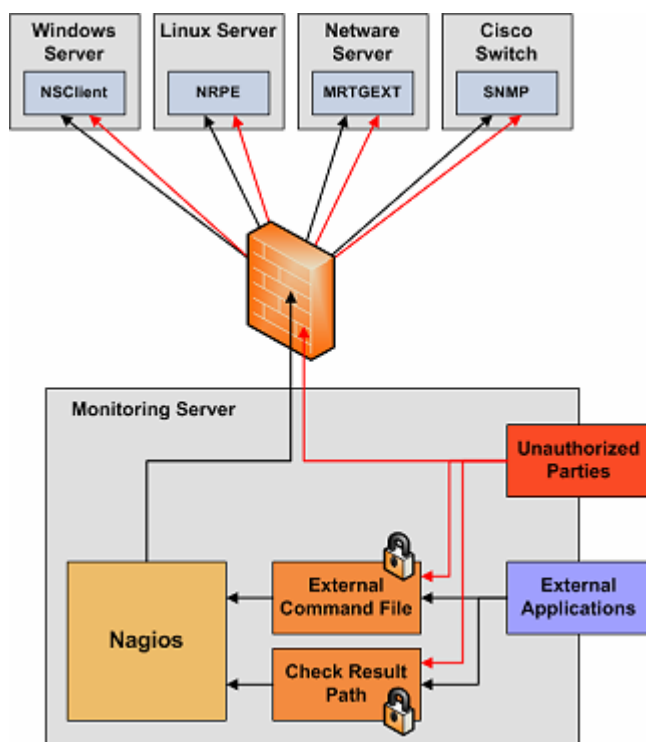
If an intruder has the ability to submit check results or external commands to the Nagios daemon, they have the potential to submit bogus monitoring data, drive you nuts you with bogus notifications, or cause event handler scripts to be triggered. If you have event handler scripts that restart services, cycle power, etc. this could be particularly problematic.

Another area of concern is the ability for intruders to sniff monitoring data (status information) as it comes across the wire. If communication channels are not encrypted, attackers can gain valuable information by watching your monitoring information. Take as an example the following situation: An attacker captures monitoring data on the wire over a period of time and analyzes the typical CPU and disk load usage of your systems, along with the number of users that are typically logged into them. The attacker is then able to determine the best time to compromise a system and use its resources (CPU, etc.) without being noticed.

Here are some tips to help ensure that you keep your systems secure when implementing a Nagios-based monitoring solution...

10.1.2. Best Practices

- **Use a Dedicated Monitoring Box.** I would recommend that you install Nagios on a server that is dedicated to monitoring (and possibly other admin tasks). Protect your monitoring server as if it were one of the most important servers on your network. Keep running services to a minimum and lock down access to it via TCP wrappers, firewalls, etc. Since the Nagios server is allowed to talk to your servers and may be able to poke through your firewalls, allowing users access to your monitoring server can be a security risk. Remember, its always easier to gain root access through a system security hole if you have a local account on a box.

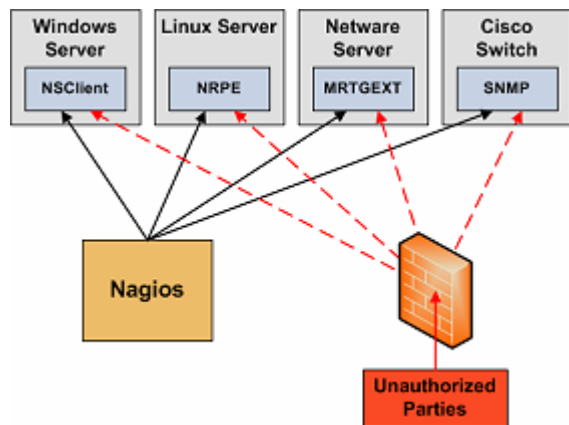


- **Don't Run Nagios As Root.** Nagios doesn't need to run as root, so don't do it. You can tell Nagios to drop privileges after startup and run as another user/group by using the [nagios_user](#) and [nagios_group](#) directives in the main config file. If you need to execute event handlers or plugins which require root access, you might want to try using [sudo](#).
- **Lock Down The Check Result Directory.** Make sure that only the **nagios** user is able to read/write in the [check result path](#). If users other than **nagios** (or **root**) are able to write to this directory, they could send fake host/service check results to the Nagios daemon. This could result in annoyances (bogus notifications) or security problems (event handlers being kicked off).

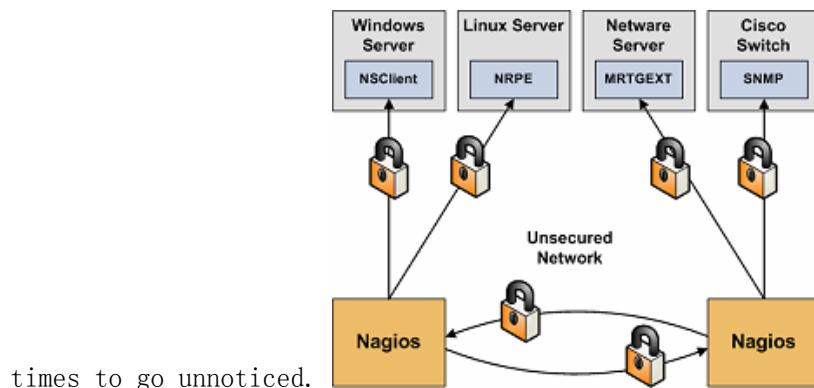
- **Lock Down The External Command File.** If you enable [external commands](#), make sure you set proper permissions on the `/usr/local/nagios/var/rw` directory. You only want the Nagios user (usually **nagios**) and the web server user (usually **nobody**, **httpd**, **apache2**, or **www-data**) to have permissions to write to the command file. If you've installed Nagios on a machine that is dedicated to monitoring and admin tasks and is not used for public accounts, that should be fine. If you've installed it on a public or multi-user machine (not recommended), allowing the web server user to have write access to the command file can be a security problem. After all, you don't want just any user on your system controlling Nagios through the external command file. In this case, I would suggest only granting write access on the command file to the **nagios** user and using something like [CGIWrap](#) to run the CGIs as the **nagios** user instead of **nobody**.
- **Require Authentication In The CGIs.** I would strongly suggest requiring authentication for accessing the CGIs. Once you do that, read the documentation on the default rights that authenticated contacts have, and only authorize specific contacts for additional rights as necessary. Instructions on setting up authentication and configuring authorization rights can be found [here](#). If you disable the CGI authentication features using the [use_authentication](#) directive in the CGI config file, the [command CGI](#) will refuse to write any commands to the [external command file](#). After all, you don't want the world to be able to control Nagios do you?
- **Use Full Paths In Command Definitions.** When you define commands, make sure you specify the **full path** (not a relative one) to any scripts or binaries you're executing.
- **Hide Sensitive Information With \$USERn\$ Macros.** The CGIs read the [main config file](#) and [object config file\(s\)](#), so you don't want to keep any sensitive information (usernames, passwords, etc) in there. If you need to specify a username and/or password in a command definition use a \$USERn\$ [macro](#) to hide it. \$USERn\$ macros are defined in one or more [resource files](#). The CGIs will not attempt to read the contents of resource files, so you can set more restrictive permissions (600 or 660) on them. See the sample **resource.cfg** file in the base of the Nagios distribution for an example of how to define \$USERn\$ macros.
- **Strip Dangerous Characters From Macros.** Use the [illegal_macro_output_chars](#) directive to strip dangerous characters from the \$HOSTOUTPUT\$, \$SERVICEOUTPUT\$, \$HOSTPERFDATA\$, and \$SERVICEPERFDATA\$ macros before they're used in notifications, etc. Dangerous characters

can be anything that might be interpreted by the shell, thereby opening a security hole. An example of this is the presence of backtick (`) characters in the \$HOSTOUTPUT\$, \$SERVICEOUTPUT\$, \$HOSTPERFDATA\$, and/or \$SERVICEPERFDATA\$ macros, which could allow an attacker to execute an arbitrary command as the nagios user (one good reason not to run Nagios as the root user).

- **Secure Access to Remote Agents.** Make sure you lock down access to agents (NRPE, NSClient, SNMP, etc.) on remote systems using firewalls, access lists, etc. You don't want everyone to be able to query your systems for status information. This information could be used by an attacker to execute remote event handler scripts or to determine the best times to go unnoticed.



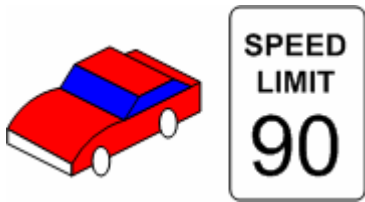
- **Secure Communication Channels.** Make sure you encrypt communication channels between different Nagios installations and between your Nagios servers and your monitoring agents whenever possible. You don't want someone to be able to sniff status information going across your network. This information could be used by an attacker to determine the best



times to go unnoticed.

10.2. Nagios 的性能调优

10.2.1. 介绍



你已经安装好 Nagios 并运行了它，并且你开始想知道如何来更好地用它。当需要监控大量(> 1,000)主机和服务有必要来优化 Nagios 以增强性能。优化 Nagios 时有几个地方要看一下...

10.2.2. 优化的招数:

- 用MRTG来绘制性能统计图表。为了保留下Nagios安装过程 In order to keep track of how well your Nagios installation handles load over time and how your configuration changes affect it, you should be graphing several important statistics with MRTG. This is really, really, really useful when it comes to tuning the performance of a Nagios installation. Really. Information on how to do this can be found [here](#).
- Use large installation tweaks. Enabling the [use large installation tweaks](#) option may provide you with better performance. Read more about what this option does [here](#).
- Disable environment macros. Macros are normally made available to check, notification, event handler, etc. commands as environment variables. This can be a problem in a large Nagios installation, as it consumes some additional memory and (more importantly) more CPU. If your scripts don't need to access the macros as environment variables (e.g. you pass all necessary macros on the command line), you don't need this feature. You can prevent macros from being made available as environment variables by using the [enable environment macros](#) option.
- Check Result Reaper Frequency. The [check result reaper frequency](#) variable determines how often Nagios should check for host and service check results that need to be processed. The maximum amount of time it can spend processing those results is determined by the max reaper time (see below). If your reaper frequency is too high (too infrequent), you might see high latencies for host and service checks.
- Max Reaper Time. The [max check result reaper time](#) variables determines the maximum amount of time the Nagios daemon can spend processing the results of host and service checks before moving on to other things – like executing new host and service checks. Too high of a value can result in large latencies for your host and service checks. Too low of a value can have the same effect. If you're experiencing high latencies, adjust this

variable and see what effect it has. Again, you should be [graphing statistics](#) in order to make this determination.

- Adjust buffer slots. You may need to adjust the value of the [external command buffer slots](#) option. Graphing buffer slot statistics with [MRTG](#) (see above) is critical in determining what values you should use for this option.
- Check service latencies to determine best value for maximum concurrent checks. Nagios can restrict the number of maximum concurrently executing service checks to the value you specify with the [max_concurrent_checks](#) option. This is good because it gives you some control over how much load Nagios will impose on your monitoring host, but it can also slow things down. If you are seeing high latency values (> 10 or 15 seconds) for the majority of your service checks (via the [extinfo CGI](#)), you are probably starving Nagios of the checks it needs. That's not Nagios's fault – it's yours. Under ideal conditions, all service checks would have a latency of 0, meaning they were executed at the exact time that they were scheduled to be executed. However, it is normal for some checks to have small latency values. I would recommend taking the minimum number of maximum concurrent checks reported when running Nagios with the `-s` command line argument and doubling it. Keep increasing it until the average check latency for your services is fairly low. More information on service check scheduling can be found [here](#).
- Use passive checks when possible. The overhead needed to process the results of [passive service checks](#) is much lower than that of "normal" active checks, so make use of that piece of info if you're monitoring a slew of services. It should be noted that passive service checks are only really useful if you have some external application doing some type of monitoring or reporting, so if you're having Nagios do all the work, this won't help things.
- Avoid using interpreted plugins. One thing that will significantly reduce the load on your monitoring host is the use of compiled (C/C++, etc.) plugins rather than interpreted script (Perl, etc) plugins. While Perl scripts and such are easy to write and work well, the fact that they are compiled/interpreted at every execution instance can significantly increase the load on your monitoring host if you have a lot of service checks. If you want to use Perl plugins, consider compiling them into true executables using `perlcc(1)`

(a utility which is part of the standard Perl distribution) or compiling Nagios with an embedded Perl interpreter (see below).

- Use the embedded Perl interpreter. If you're using a lot of Perl scripts for service checks, etc., you will probably find that compiling the [embedded Perl interpreter](#) into the Nagios binary will speed things up.
- Optimize host check commands. If you're checking host states using the `check_ping` plugin you'll find that host checks will be performed much faster if you break up the checks. Instead of specifying a **max_attempts** value of 1 in the host definition and having the `check_ping` plugin send 10 ICMP packets to the host, it would be much faster to set the **max_attempts** value to 10 and only send out 1 ICMP packet each time. This is due to the fact that Nagios can often determine the status of a host after executing the plugin once, so you want to make the first check as fast as possible. This method does have its pitfalls in some situations (i.e. hosts that are slow to respond may be assumed to be down), but I you'll see faster host checks if you use it. Another option would be to use a faster plugin (i.e. `check_fping`) as the **host_check_command** instead of `check_ping`.
- Schedule regular host checks. Scheduling regular checks of hosts can actually help performance in Nagios. This is due to the way the [cached check logic](#) works (see below). Prior to Nagios 3, regularly scheduled host checks used to result in a big performance hit. This is no longer the case, as host checks are run in parallel – just like service checks. To schedule regular checks of a host, set the **check_interval** directive in the [host definition](#) to something greater than 0.
- Enable cached host checks. Beginning in Nagios 3, on-demand host checks can benefit from caching. On-demand host checks are performed whenever Nagios detects a service state change. These on-demand checks are executed because Nagios wants to know if the host associated with the service changed state. By enabling cached host checks, you can optimize performance. In some cases, Nagios may be able to use the old/cached state of the host, rather than actually executing a host check command. This can speed things up and reduce load on monitoring server. In order for cached checks to be effective, you need to schedule regular checks of your hosts (see above). More information on cached checks can be found [here](#).

- Don't use aggressive host checking. Unless you're having problems with Nagios recognizing host recoveries, I would recommend not enabling the [use aggressive host checking](#) option. With this option turned off host checks will execute much faster, resulting in speedier processing of service check results. However, host recoveries can be missed under certain circumstances when this is turned off. For example, if a host recovers and all of the services associated with that host stay in non-OK states (and don't "wobble" between different non-OK states), Nagios may miss the fact that the host has recovered. A few people may need to enable this option, but the majority don't and I would recommend not using it unless you find it necessary...
- External command optimizations. If you're processing a lot of external commands (i.e. passive checks in a [distributed setup](#), you'll probably want to set the [command check interval](#) variable to -1. This will cause Nagios to check for external commands as often as possible. You should also consider increasing the number of available [external command buffer slots](#). Buffer slots are used to hold external commands that have been read from the [external command file](#) (by a separate thread) before they are processed by the Nagios daemon. If your Nagios daemon is receiving a lot of passive checks or external commands, you could end up in a situation where the buffers are always full. This results in child processes (external scripts, NSCA daemon, etc.) blocking when they attempt to write to the external command file. I would highly recommend that you graph external command buffer slot usage using MRTG and the nagiosstats utility as described [here](#), so you understand the typical external command buffer usage of your Nagios installation.
- Optimize hardware for maximum performance. NOTE: Hardware performance shouldn't be an issue unless: 1) you're monitoring thousands of services, 2) you're doing a lot of post-processing of performance data, etc. Your system configuration and your hardware setup are going to directly affect how your operating system performs, so they'll affect how Nagios performs. The most common hardware optimization you can make is with your hard drives. CPU and memory speed are obviously factors that affect performance, but disk access is going to be your biggest bottleneck. Don't store plugins, the status log, etc on slow drives (i.e. old IDE drives or NFS mounts). If you've got them, use UltraSCSI drives or fast IDE drives. An important note for IDE/Linux users is that many Linux installations do not attempt to optimize disk access. If you don't change the disk access parameters

(by using a utility like `hdparam`), you'll loose out on a lot of the speedy features of the new IDE drives.

10.3. 使用 Nagios 状态工具

10.3.1. 介绍

在Nagios发行包中含有一个名为`nagiosstats`的工具，它与Nagios主程序一起被编译和安装。Nagios状态工具可以在线地收集各种Nagios的运行信息并将在[性能调优](#)中非常有用。可以把信息搞成要么是可读的要么MRTG兼容型的格式。

10.3.2. 用法信息

可以用参数`-help`来运行 **nagiosstats** 以取得用法信息。

10.3.3. 可阅读的输出

为获取人可阅读的在线运行 Nagios 性能数据的信息，使用命令行`-c` 参数来运行 **nagiosstats** 工具并指定主配置文件位置，象这样：

```
[nagios@lanman ~]# /usr/local/nagios/bin/nagiosstats -c /usr/local/nagios/etc/nagios.cfg
```

```
Nagios Stats 3.0prealpha-05202006
```

```
Copyright (c) 2003-2007 Ethan Galstad (www.nagios.org)
```

```
Last Modified: 05-20-2006
```

```
License: GPL
```

```
CURRENT STATUS DATA
```

```
-----  
Status File: /usr/local/nagios/var/status.dat
```

```
Status File Age: 0d 0h 0m 9s
```

```
Status File Version: 3.0prealpha-05202006
```

```
Program Running Time: 0d 5h 20m 39s
```

```
Nagios PID: 10119
```

```
Used/High/Total Command Buffers: 0 / 0 / 64
```

```
Used/High/Total Check Result Buffers: 0 / 7 / 512
```

Total Services:	95
Services Checked:	94
Services Scheduled:	91
Services Actively Checked:	94
Services Passively Checked:	1
Total Service State Change:	0.000 / 78.950 / 1.026 %
Active Service Latency:	0.000 / 4.272 / 0.561 sec
Active Service Execution Time:	0.000 / 60.007 / 2.066 sec
Active Service State Change:	0.000 / 78.950 / 1.037 %
Active Services Last 1/5/15/60 min:	4 / 68 / 91 / 91
Passive Service State Change:	0.000 / 0.000 / 0.000 %
Passive Services Last 1/5/15/60 min:	0 / 0 / 0 / 0
Services Ok/Warn/Unk/Crit:	58 / 16 / 0 / 21
Services Flapping:	1
Services In Downtime:	0
Total Hosts:	24
Hosts Checked:	24
Hosts Scheduled:	24
Hosts Actively Checked:	24
Host Passively Checked:	0
Total Host State Change:	0.000 / 9.210 / 0.384 %
Active Host Latency:	0.000 / 0.446 / 0.219 sec
Active Host Execution Time:	1.019 / 10.034 / 2.764 sec
Active Host State Change:	0.000 / 9.210 / 0.384 %
Active Hosts Last 1/5/15/60 min:	5 / 22 / 24 / 24
Passive Host State Change:	0.000 / 0.000 / 0.000 %
Passive Hosts Last 1/5/15/60 min:	0 / 0 / 0 / 0
Hosts Up/Down/Unreach:	18 / 4 / 2
Hosts Flapping:	0
Hosts In Downtime:	0

```

Active Host Checks Last 1/5/15 min:    9 / 52 / 164
Scheduled:                             4 / 23 / 75
On-demand:                             3 / 23 / 69
Cached:                                2 / 6 / 20
Passive Host Checks Last 1/5/15 min:    0 / 0 / 0
Active Service Checks Last 1/5/15 min:  9 / 80 / 244
Scheduled:                             9 / 80 / 244
On-demand:                             0 / 0 / 0
Cached:                                0 / 0 / 0
Passive Service Checks Last 1/5/15 min: 0 / 0 / 0
External Commands Last 1/5/15 min:     0 / 0 / 0
[nagios@lanman ~]#

```

如你所见，它显示了 Nagios 进程在不同统计频度上的一系列数字，有多个值在统计频度上显示，主要是(除非特别指定)最小值、最大值和平均值。

10.3.4. MRTG 集成

可以将 **nagiosstats** 工具与 MRTG 或其他兼容程序集成来显示 Nagios 的统计结果。为完成它，用 `--mrtg` 和 `--data` 参数来运行 **nagiosstats** 工具。参数 `--data` 可指定哪个哪种统计值被绘制成图，可用的值可以通过用 `--help` 命令参数运行 **nagiosstats** 来查找。



注意：有关使用 **nagiosstats** 来对 Nagios 统计状态结果绘制 MRTG 图表信息可以查阅 [这篇文档](#)。

10.4. 使用 MRTG 绘制性能数据

10.4.1. 介绍

[Nagios 状态应用工具](#) 可以利用 [MRTG](#) 绘制多种 Nagios 性能统计图表。这个很重要，因为它可以：

- 确保 Nagios 被更有效率地操作；
- 定位监控过程中的问题；
- 感知因 Nagios 配置修改而导致的性能冲突影响；

10.4.2. MRTG 配置样例

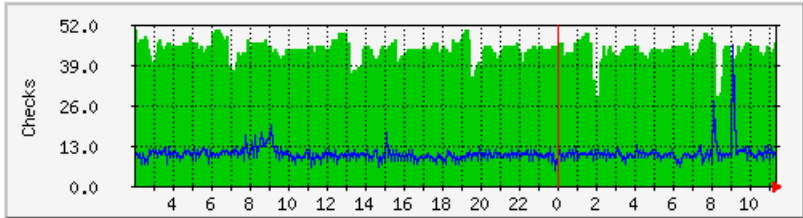
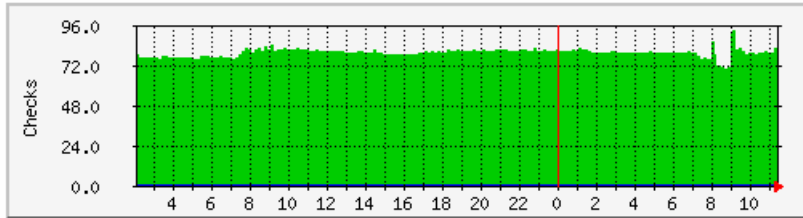
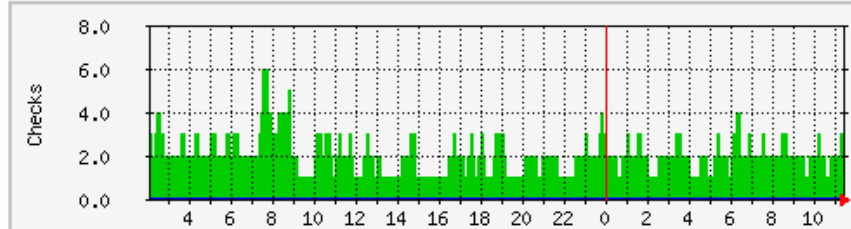
绘制各种 Nagios 的性能统计图 MRTG 配置文件片段可查看 Nagios 发行包里 **sample-config/** 子目录下的 **mrtg.cfg** 文件。如果需要可以创建性能信息的其他图表文件 — 样例只是提供了一个好的起点。

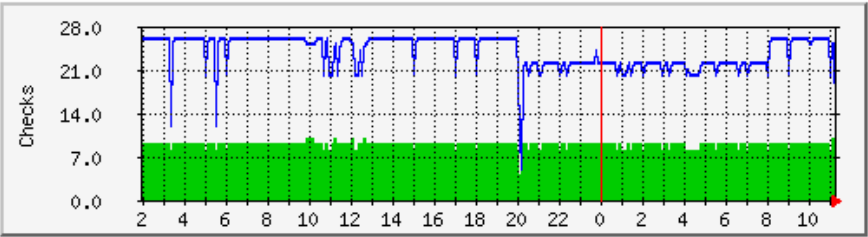
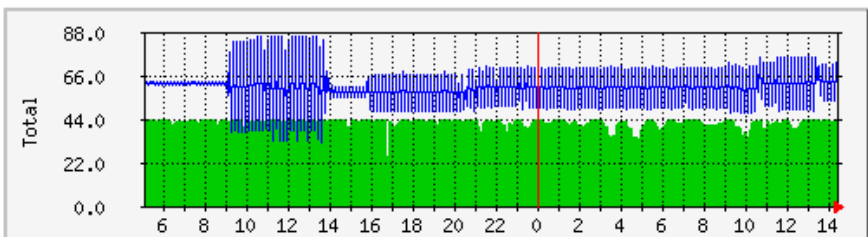
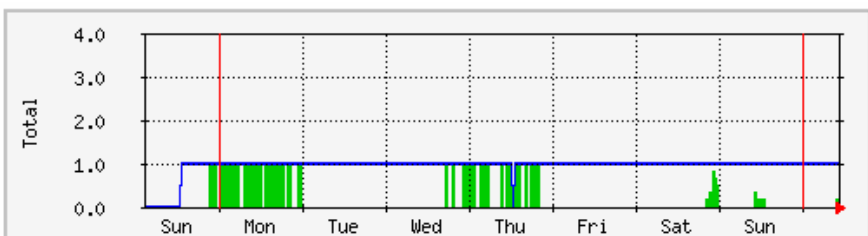
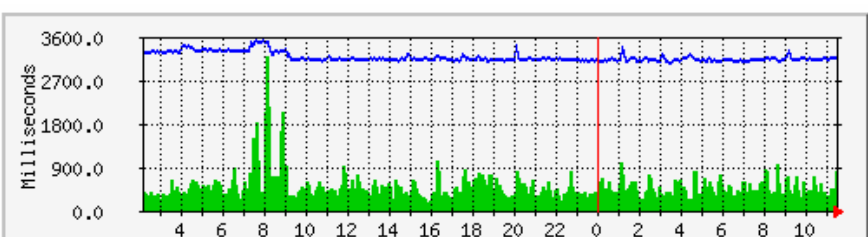
一旦你复制这些样例文件到你的 MRTG 配置文件 (/etc/mrtg/mrtg.cfg) 里, 你将在 MRTG 的下次运行时得到这些新图表。

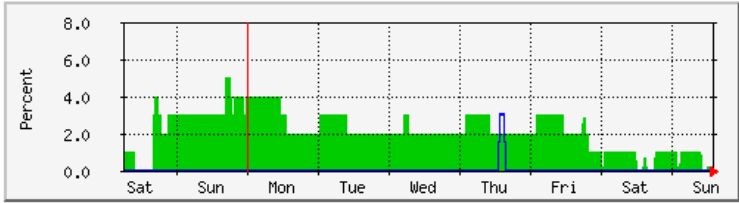
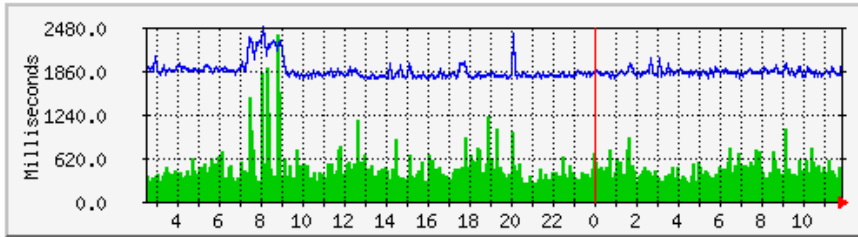
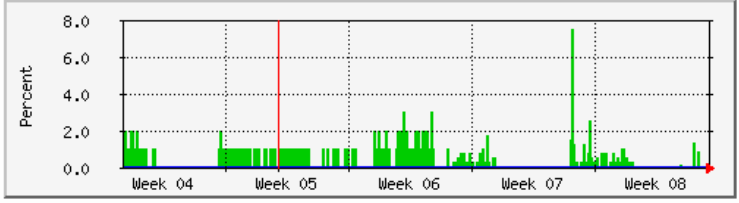
10.4.3. 图表实例

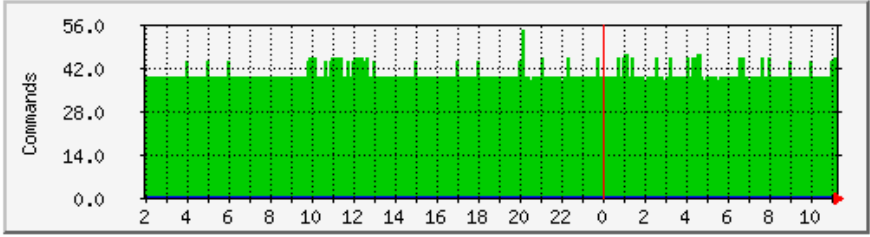
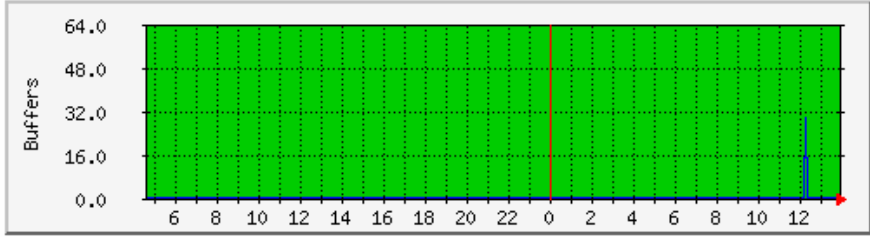
下面将描述一下几个样例 MRTG 图表的内容及用途...

表 10.1.

<p>自主主机检测—该图显示了沿时间轴做过多少次自主主机检测(包括规格化计划检测和按需检测), 有助于理解:</p> <ul style="list-style-type: none">• 主机检测• 主机依赖检测的前处理• 缓存检测	 <p>Max Scheduled Checks: 50.0 Average Scheduled Checks: 44.0 Current Scheduled Checks: 46.4 Max On-Demand Checks: 45.0 Average On-Demand Checks: 10.0 Current On-Demand Checks: 10.0</p>
<p>自主服务检测—该图显示了沿时间轴做过多少次自主服务检测(包括规格化计划检测和按需检测), 有助于理解:</p> <ul style="list-style-type: none">• 服务检测• 服务依赖检测的前处理• 缓存检测	 <p>Max Scheduled Checks: 93.0 Average Scheduled Checks: 72.0 Current Scheduled Checks: 66.4 Max On-Demand Checks: 0.0 Average On-Demand Checks: 0.0 Current On-Demand Checks: 0.0</p>
<p>主机和服务检测缓存检测—该图显示了沿时间轴做过多少次主机与服务缓存检测。有助于理解:</p> <ul style="list-style-type: none">• 缓存检测• 主机与服务依赖检查的前处理	 <p>Max Host Checks: 6.0 Average Host Checks: 2.0 Current Host Checks: 3.0 Max Service Checks: 0.0 Average Service Checks: 0.0 Current Service Checks: 0.0</p>

<p>强制主机和服务检测—该图显示了沿时间轴做过多少次强制主机与服务检测。有助于理解：</p> <ul style="list-style-type: none"> 强制检测 	 <p>Max Host Checks: 10.0 Average Host Checks: 9.0 Current Host Checks: 6.0 Max Service Checks: 26.0 Average Service Checks: 24.0 Current Service Checks: 12.0</p>
<p>主机和服务自主检测—该图显示了沿时间轴上持续地有多少个主机与服务(总数量)自主检测。有助于理解：</p> <ul style="list-style-type: none"> 自主检测 	 <p>Max Hosts: 44.0 Average Hosts: 42.0 Current Hosts: 43.0 Max Services: 85.0 Average Services: 60.0 Current Services: 53.0</p>
<p>主机和服务强制检测—该图显示了沿时间轴上持续地有多少个主机与服务(总数量)强制检测。有助于理解：</p> <ul style="list-style-type: none"> 强制检测 	 <p>Max Hosts: 1.0 Average Hosts: 0.0 Current Hosts: 0.0 Max Services: 1.0 Average Services: 1.0 Current Services: 1.0</p>
<p>服务检测传输时延与执行的平均时间—该图显示了沿时间轴上有关服务检测的传输时延和执行时间的平均值。有助于理解：</p> <ul style="list-style-type: none"> 服务检测 性能调优 <p>若有居高不下传输时延可能是由于下列参数需要调整：</p> <ul style="list-style-type: none"> max_concurrent_checks check_result_reaper_frequency max_check_result_reap 	 <p>Max Latency: 3203.0 Average Latency: 463.0 Current Latency: 816.0 Max Execution Time: 3498.0 Average Execution Time: 3163.0 Current Execution Time: 3185</p>

<p>er_time</p>	
<p>服务状态改变的平均值—该图显示了沿时间轴服务状态改变的百分比(变化率的度量)，不论是在自主还是强制方式，最后一次检测显示服务中止的情况。</p> <p>有助于理解：</p> <ul style="list-style-type: none">• 抖动检查	 <p>Max Active Check % Change: 5.0 Average Active Check % Change: 2.0 Current Active Check % Change: 3.0 Max Passive Check % Change: 3.0 Average Passive Check % Change: 0.0 Current Passive Check % Change: 0.0</p>
<p>主机检测的传输时延与执行的平均时间—该图显示了沿时间轴上主机检测传输时延和执行时间的平均值。有助于理解：</p> <ul style="list-style-type: none">• 主机检测• 性能调优 <p>若有居高不下的传输时延可能需要调整下列参数：</p> <ul style="list-style-type: none">• max_concurrent_checks• check_result_reaper_frequency• max_check_result_reaper_time	 <p>Max Latency: 2373.0 Average Latency: 433.0 Current Latency: 374.0 Max Execution Time: 2480.0 Average Execution Time: 1839.0 Current Execution Time: 2021</p>
<p>平均主机状态改变—该图显示了沿时间轴主机状态发生变化的百分比(变化率的度量)，不论是在自主还是强制检测方式，最后一次主机检测的中止情况。有助于理解：</p> <ul style="list-style-type: none">• 抖动检查	 <p>Max Active Check % Change: 7.0 Average Active Check % Change: 1.0 Current Active Check % Change: 3.0 Max Passive Check % Change: 0.0 Average Passive Check % Change: 0.0 Current Passive Check % Change: 0.0</p>

<p>外部命令—该图显示了 Nagio 主守护进程沿时间轴有多少个外部命令要处理。除非要处理大量的外部命令(如在分布式安装环境下),该图基本上是空白的。监视外部命令将有助于如下内容的影响理解:</p> <ul style="list-style-type: none">• 强制检测• 分布式监控• 冗余和失效监控	 <p>Max Commands: 34.0 Average Commands: 40.0 Current Commands: 20.0</p>
<p>外部命令缓冲—该图显示了沿时间轴多少外部命令使用缓冲。如果使用中的缓冲数量接近了可用缓冲数量,说明需要增加可用的 外部命令缓冲块。每个缓冲块可存放一个外部命令。缓冲被用于临时存入外部文件,临时期开始于外部命令自 外部命令文件中取出时刻,结束于Nagios守护程序处理完成外部命令结果。</p>	 <p>Max Avail: 64.0 Average Avail: 64.0 Current Avail: 64.0 Max Used: 29.0 Average Used: 0.0 Current Used: 0.0</p>

10.5. 对 CGI 程序模块的授权与认证

10.5.1. 介绍

本文给出了 Nagios 的 CGI 程序模块如何确定授权权限以对如下操作付权: 获取监控数据、配置信息和通过用 WEB 接口对 Nagios 守护程序发出指令。

10.5.2. 定义

在此之前, 很重要的一点是要理解联系人授权与认证的含义及两者的不同之处, 主要有:

- 一个认证用户是指有权以指定的用户名和口令通过 WEB 服务器认证可以获取 Nagios 的 WEB 接口页面的用户;
- 一个认证的联系人是在 [联系人定义](#)文件里的短用户名中列出的用户名;

10.5.3. 设置认证用户

假定你已经按 [快速指南](#)里的方式配置好Web服务器，在调用Nagios的CGI程序模块前要认证。而且已经有了一个用户帐号(**nagiosadmin**)或对CGI模块操作。

想定义更多的 [联系人](#)来接收主机和服务的通知，一般是想让联系人通过Nagios的Web接口来做。可以按下面命令来给CGI程序指定额外的用户，把下面<username>用真实的想加入的用户名来替换。一般情况下，这个名字应是和配置文件中的 [联系人](#)对象定义中的短名称相匹配。

```
htpasswd /usr/local/nagios/etc/htpasswd.users <username>
```

10.5.4. 打开 CGI 模块的认证与授权功能

下一步是配置CGI模块使用认证与授权功能来决定什么样的信息或是命令可以操作。把 [CGI配置文件](#) 里面的 [use_authentication](#)选项置为非零值，如：

```
use_authentication=1
```

好了，设置好 CGI 模块的认证与授权了。

10.5.5. 给 CGI 模块的默认许可权限

当使能了认证与授权功能后，CGI 模块将给用户什么默认许可呢？

表 10.2. 默认许可权限

CGI 模块的数据	认证的联系人*	其他认证的用户(非设定的对象联系人)*
主机状态信息	Yes	No
主机配置信息	Yes	No
主机历史	Yes	No
主机通知	Yes	No
主机命令	Yes	No
服务状态信息	Yes	No
服务配置信息	Yes	No
服务历史	Yes	No
服务通知	Yes	No
服务命令	Yes	No

CGI 模块的数据	认证的联系人 *	其他认证的用户 (非设定的对象联系人) *
全部配置信息	No	No
系统/进程信息	No	No
系统/进程命令	No	No

认证的联系人 [*](#)可以取得每个以他为联系人的每个服务 (联系人不是他的服务不行)...

- 授权查看服务状态信息;
- 授权查看服务配置信息;
- 授权查看该服务的历史与通知;
- 授权发出服务命令

认证的联系人 [*](#)可以对每个以他为联系人的每个主机 (联系人不是他的主机不行)...

- 授权查看主机状态信息;
- 授权查看主机配置信息;
- 授权查看主机的历史和通知;
- 授权发出主机命令;
- 授权查看在该主机上全部服务的状态信息;
- 授权查看在该主机上全部服务的配置信息;
- 授权查看在该主机上全部服务的历史与通知;
- 授权给在该主机上全部服务发出命令。

重要一点是默认情况下以下内容无人被授权得到如下内容...

- 用 [日志查看CGI](#)查看打包的日志文件;
- 用 [扩展信息CGI](#)查看Nagios进程的信息;
- 用 [命令CGI](#)对Nagios进程发出命令;
- 用 [配置CGI](#)查看主机组、联系人、联系人组、时间周期和命令定义;

毫无疑问需要这些信息, 所以要看下面的内容以使你 (可能包括其他人) 有权限得到这些额外的信息...

10.5.6. 给 CGI 增加额外的权限

可以允许让**认证的联系人**或其他**认证的用户**有权限得到CGI模块里的额外信息, 通过在 [CGI配置文件](#)里增加一些授权变量来实现。我实现了如下的授权变量以使他们可以控制无授权的情况下不能取得信息, 总归比没有这些要好吧...

在 CGI 配置文件里加了如下的变量控制额外的授权内容...

- [authorized for system information](#)
- [authorized for system commands](#)
- [authorized for configuration information](#)
- [authorized for all hosts](#)
- [authorized for all host commands](#)
- [authorized for all services](#)
- [authorized for all service commands](#)

10.5.7. CGI 模块的授权要求

如果被各种各样的CGI模块里所需要的授权搞糊涂了,可以看一下每个CGI模块介绍里所写的**授权要求**的说明,在 [这个](#)文档里面。

10.5.8. 在加密的 Web 服务器上认证

如果WEB服务器是建在一个加密域(象在防火墙后面)或是用SSL加密通讯的,可以设置一个默认用户来完成CGI操作。可以在 [CGI配置文件](#)里设置 [default_user_name](#)选项。通过设置一个默认的用户来操作CGI模块,可以不必再由WEB服务器来做验证。这样通过因特网,可以省去基本的WEB认证过程,或以空白的口令进行基本认证(基本数据已经加过密了)。

Important:不要使用默认的用户名这个功能,除非运行在个加密的 Web 服务或加密域里,每个人都经过了充分认证后才会操作 CGI 模块,因为此时没有经过 Web 认证的每一个用户都具备相同的、全部的设置操作权限!

10.6. 用户定制 CGI 页面头和尾

10.6.1. 介绍

如果你安装了Nagios的客户端,你可能需要定制自己的 [CGI模块](#)的页面头和尾以显示自己的信息。这对于向最终用户提供联系人信息等内容时比较有用。

很重要一点是,除非它们会被执行,否则用户自定制的页面头和尾的内容在显示之前不会被预先处理。页面的头和尾内容包含文件只是被简单地读入并显示到 CGI 页面上,这意味着头和尾的定义中只能包含一些特定的可被浏览器支持的内容(如 HTML、JavaScript 等)。

如果用户定制的头和尾文件是可执行的,它们会在 CGI 模块被调用时显示到最终用户的浏览器,因而它们必须是合法的 HTML 结果。利用这一点可以执行你定制的 CGI 程序来插入到 Nagios 的显示数据之中,这已被用于从 RRDTOOL 中获取的图表(用 ddraw 命令)或是在 Nagios 的显示板上显示命令菜单。可执行的用户定制的 CGI 页面的头和尾与 Nagios 的 CGI 程序使用相同的 CGI 环境,因而你的 CGI 程序可以同样地解析 URL 调用行信息、用户验证信息等以制作出你的输出内容。

10.6.2. 它是如何工作的？

你可以把 CGI 程序模块所包含的用户定制头尾内容，经常是 HTML 文件放在 Nagios 的 HTML 的 `ssi/` 子目录中，一般是在这个位置 `/usr/local/nagios/share/ssi` 上。

用户定制头通常是紧跟在 CGI 的 `<BODY>` 标记之后而尾经常是跟在 `</BODY>` 标记之前。

有两种类型的用户定制的头和尾：

- 全局的头和尾定义。文件必须分别被命名为 `common-header.ssi` 和 `common-footer.ssi`。如果它们被定义，它们将被全部的 CGI 程序模块所调用显示。

- 特定的 CGI 模块的用户定制的头和尾。文件必须被命名为如下格式 `CGINAME-header.ssi` 和

`CGINAME-footer.ssi`，这里的 `CGINAME` 是那个 CGI 程序模块不带有 `.cgi` 后缀的部分。比如给 [报警汇总 CGI 模块](#) 即 (`summary.cgi`) 的头和尾必须分别被命名为 `summary-header.ssi` 和

`summary-footer.ssi`。

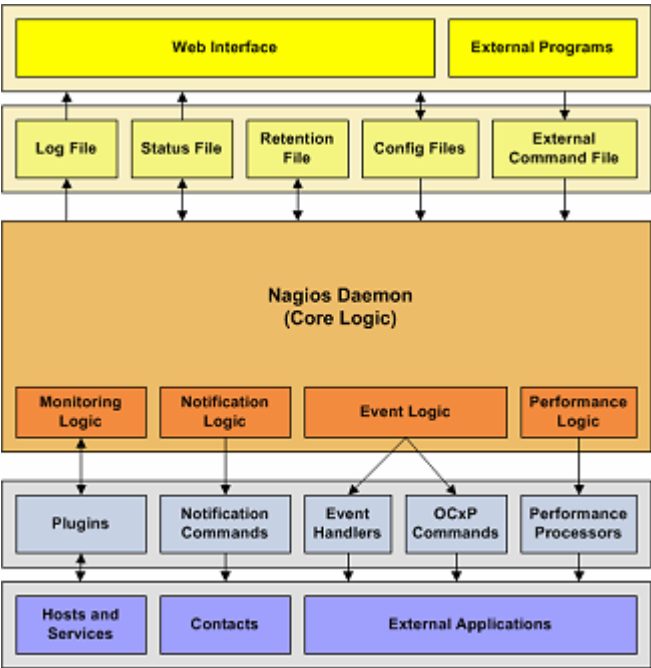
如果你并不需要用户定制的头和尾，你可以只用全局的头定义或是全局的尾定义，真的要看你所需了。

第 11 章 软件集成相关的内容

第 11 章 软件集成相关的内容

11.1. 软件集成概览

11.1.1. 介绍



Nagios 可以非常容易地与现有框架集成，这也就是为何 Nagios 被广泛地应用的一原因。有不少方式来与现有管理软件进行集成，你使用管理软件来监控你所拥有的各种各样的新型或用户定制的硬件、服务或是应用程序。

11.1.2. 集成的要点

为了监控新硬件、服务或是应用程序，审视如下的文档：

- [Nagios插件](#)
- [插件API](#)
- [强制检测](#)
- [事件处理句柄](#)

为使 Nagios 取得外部应用程序的数据，审视如下的文档：

- [强制检测](#)
- [外部命令](#)

将状态、性能或是告警信息报送给外部应用，审视如下文档：

- [事件处理句柄](#)
- [OCSP](#)和 [OCHP](#)命令
- [性能数据](#)
- [告警](#)

11.1.3. 集成事例

我记录下了一些事例来看一下 Nagios 是如何与外部程序集成的，它们是：

- [TCP Wrappers](#) (安全事件报警)
- [SNMP Traps](#) (卷备份作业的状态)

11.2. SNMP 陷阱集成

11.2.1. 介绍

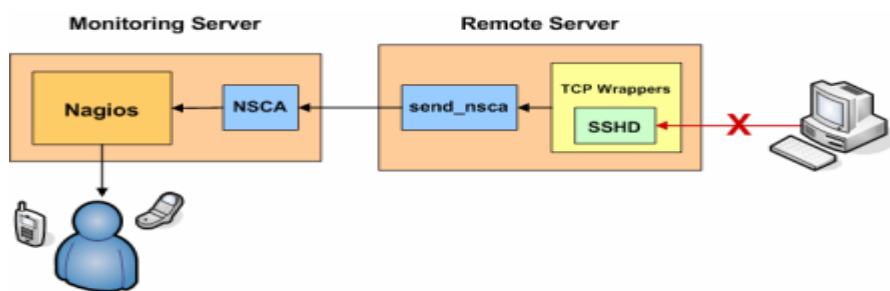
注意



Nagios并没有设计成一个可替代完全SNMP管理功能的象HP OpenView或 [OpenNMS](#)那样的应用程序。然而，你可以在Nagios中设置好SNMP陷阱来接收来自于网络中的主机发出的SNMP警报。

SNMP的无所不管除了恶长以外一无是处。接收SNMP消息并将它放到Nagios里(象强制检测结果一样)是件很繁闷的事。为使之更简单，建议你取出Alex Burger的SNMP Trap Translator项目，它位于<http://www.snmpTT.org/>，这里面在Nagios里集成了Net-SNMP、SNMP TT及增强型的消息陷阱处理系统。

11.3. TCP Wrapper 集成



11.3.1. 介绍

本文档解释如何容易地在 Nagios 里用 TCP Wrapper 对联接尝试被拒绝而产生警报。例如，一个非授权主机试图联接 SSH 服务器，可以在 Nagios 里收到一个含有被拒绝主机名的警报。如果在 Linux/Unix 机器上实现了这个功能，就会惊讶地发现在网络里竟会有如此多的端口扫描。

集成的前提是：

- 已经熟悉 [强制检测](#) 及其工作方式；
- 已经熟悉 [可变服务](#) 及其工作方式；
- 想要报警的主机(就是使用了 TCP wrappers 的机器)是一台远程主机(在例子中命名为 **firestorm**)。如果与运行 Nagios 的机器是同一台，需要对下面给出的例子做些修改才行。
- 已经在 Nagios 监控服务机上安装有 [NSCA 守护服务](#) 并且在那台安装有 TCP Wrapper 的远程主机上安装有 NSCA (**send_nsca**) 客户端软件。

11.3.2. 定义一个服务

如果条件具备，给远程主机 (**firestorm**) 创建一个 [主机对象定义](#)。

下一步，给这台运行有 TCP Wrapper 的主机 (**firestorm**) 在 [对象配置文件](#) 里加一个服务对象定义，服务对象定义的可能会有这样的：

```
define service{
    host_name                firestorm
    service_description      TCP Wrappers
    is_volatile               1
    active_checks_enabled    0
    passive_checks_enabled   1
    max_check_attempts        1
    check_command              check_none
    ...
}
```

```
}
```

在服务对象定义里有几个很重要：

- 打开**可变服务**(`is_volatile=1`)功能开关，因为每一个检测到的报警都要送出一个通知；
- 服务的自主检测被关闭，而强制检测功能打开。这说明 Nagios 将不会对服务自主地做检测—全部报警信息将是由外部源通过强制方式提供给 Nagios；
- 服务对象里的 `max_check_attempts` 值设定为 1。这保证了当首条报警产生时就有送出一个通知。

11.3.3. 配置 TCP Wrappers

现在需要修改在 **firestorm** 机器上的 `/etc/hosts.deny` 文件了。为使 TCP wrappers 对每个被拒绝的联接尝试都送出一条报警，需要加上这一行：

```
ALL: ALL: RFC931: twist (/usr/local/nagios/libexec/eventhandlers/handle_tcp_wrapper %h %d) &
```

这行里假定在 **firestorm** 机器上有个脚本名字是 `handle_tcp_wrapper` 且放在 `/usr/local/nagios/libexec/eventhandlers/`目录下，下面会给出脚本内容。

11.3.4. 写那个脚本

最后一件事是在 **firestorm** 上写那个 `handle_tcp_wrapper` 脚本，它将会把报警送给 Nagios 监控服务器，它可能会是这样的：

```
#!/bin/sh

/usr/local/nagios/libexec/eventhandlers/submit_check_result firestorm "TCP Wrappers" 2
"Denied $2-$1" > /dev/null 2> /dev/null
```

注意 `handle_tcp_wrapper` 脚本调用了 `submit_check_result` 脚本来真正地送出报警。假定 Nagios 服务器被命名为 **monitor**，这个 `submit check_result` 脚本内容可能会是这样的：

```
#!/bin/sh

# Arguments

# $1 = name of host in service definition
# $2 = name/description of service in service definition
# $3 = return code
# $4 = output

/bin/echo -e "$1\t$2\t$3\t$4\n" | /usr/local/nagios/bin/send_nscs monitor -c
/usr/local/nagios/etc/send_nscs.cfg
```


11.3.5. 搞好了

已经全配置完成了，可以重新启动一下 **firestorm** 机器上的 **inetd** 进程，并且重新启动一下监控服务器上的 Nagios 进程。搞定了。当 **firestorm** 上的 TCP wrappers 拒绝了一次联接尝试时，将会在 Nagios 里看到一条报警。报警的插件输出将会是这样的：

```
Denied sshd2-sdn-ar-002mnminnP321.dialsprint.net
```

11.4. Nagios 外部构件

11.4.1. 介绍

Nagios 有许多“外部构件”软件包可供使用。外部构件可以扩展 Nagios 的应用并使之与其他软件集成。

外部构件可用于：

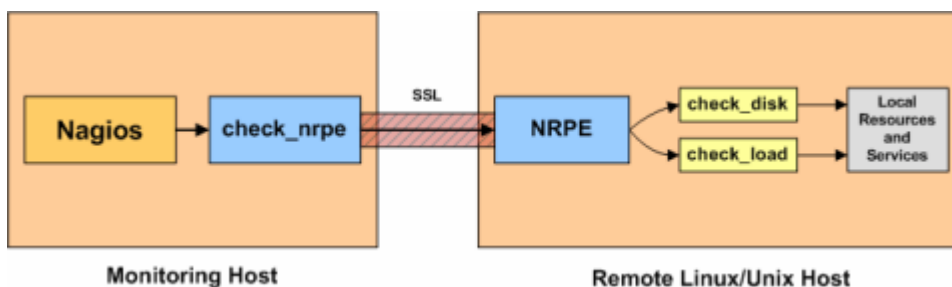
- 通过 WEB 接口来管理配置文件
- 监控远程主机(*NIX, Windows, 等)
- 实现对远程主机的强制检测
- 减化并扩展告警逻辑
- ... 和其他更多事情

你可以通过访问如下站点找寻外部构件：

- Nagios.org
- SourceForge.net
- NagiosExchange.org

这里对一些我开发的外部构件给一个简洁的介绍...

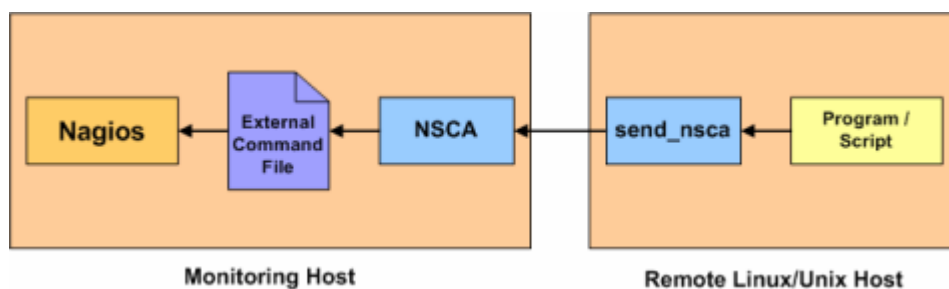
11.4.2. NRPE



NRPE是一个可在远程Linux/Unix主机上执行的 插件 的外部构件包。如果你需要监控远程的主机上的本地资源或属性，如磁盘利用率、CPU负荷、内存利用率等时是很有用的。象是用 **check_by_ssh** 插件来实现的功能一样，但是它不需要占用更多的监控主机的CPU负荷—当你需要监控成百上千个主机是这个很重要。

NRPE外部构件包和文档可以在 <http://www.nagios.org/> 上找到。

11.4.3. NSCA

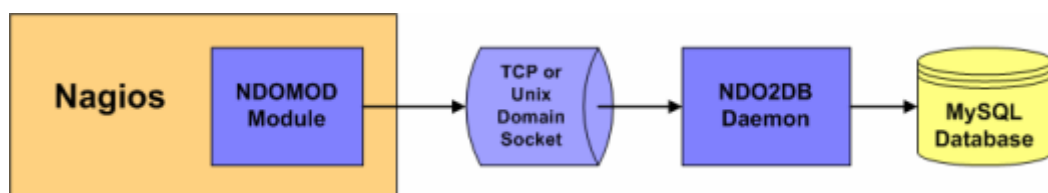


NSCA是一个可在远程Linux/Unix主机上执行 [强制检测](#)并将结果传给Nagios守护进程的外部构件包。

这在 [分布式](#)和 [冗余/失效](#)监控的设置时非常有用。

NSCA外部构件包和文档可以在 <http://www.nagios.org/>上找到。

11.4.4. NDOUtils



NDOUtils 是一个可以把全部状态信息保存到 MySQL 数据库里的外部构件。外个 Nagios 的库实例都可以把它们监控的信息保存到统一的中心数据库并集中报告。它将为一个 Nagios 新的基于 PHH 的 WEB 接口程序提供数据源服务。

NDOUtils外部构件包和文档可以在 <http://www.nagios.org/>上找到。

第 12 章 开发相关

第 12 章 开发相关

12.1. 使用内嵌 Perl 解释器

12.1.1. 介绍

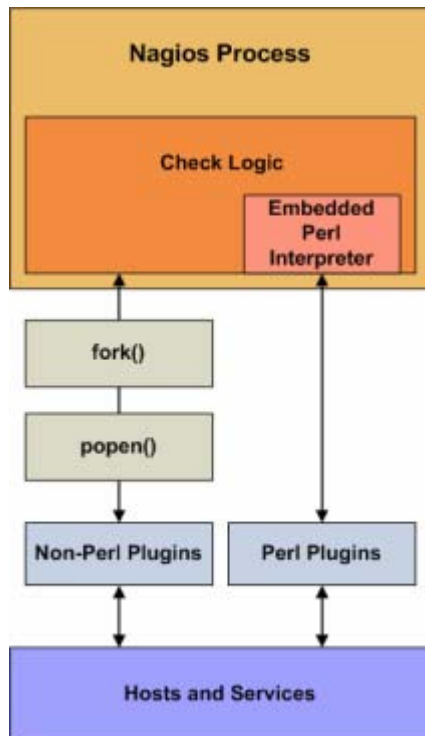
Nagios 编译时可以选择支持内嵌 Perl 解释器。这使得 Nagios 可以用更高效率来执行 Perl 所写插件，因而如果严重依赖于 Perl 写的插件的话可能是个好消息。没有内嵌 Perl 解释器，Nagios 将通过象外部命令一样用派生和执行的方法利用 Perl 所写的插件。当编译中选择了支持内嵌 Perl 解释器时，Nagios 可以象调用库一样来执行 Perl 插件。

提示



嵌入式 Perl 解释器可以让 Nagios 执行各种 Perl 脚本——不仅仅是插件。本文档只讨论涉及到 Perl 解释里执行对主机与服务检测的插件，但它同样也支持相似的 Perl 脚本以用于其他类型命令(例如通知脚本、事件处理脚本等)。

Stephen Davies 在几年前发布了最初的嵌入式 Perl 解释器，Stanley Hopcroft 是主要的帮助嵌入式 Perl 解释器改进提升的人并且应用它的对优劣性做了批注。他同时给出了有关如何更好地实现用嵌入式 Perl 来实现 Perl 插件的方法。必须注意本文档里用的“ePN”，它指示 Nagios 用 Perl，或是指示了 Nagios 要用嵌入式 Perl 解释器来编译执行它。



12.1.2. 优点

使用 ePN(Nagios 编译有嵌入式 Perl 解释器)的好处有：

- Nagios 在运行 Perl 插件时将付出更少时间因为它不需要派生进程来执行插件(每次执行要调入 Perl 解释器)。嵌入式 Perl 解释器可以象调用库函数一样来执行插件；
- 它会大大降低运行 Perl 插件的系统开销并且(或者同时)可以同时运行更多的 Perl 插件检测。也就是说，可能使用其他语言并没有这些好处，语言象 C/C++、Expect/TCL，用这些语言开发插件公认地比使用 Perl 语言开发插件要慢一个数量级(虽然在最终运行时间上会更快，TCL 语言是个例外)；
- 如果不是 C 程序员，仍旧可以用 Perl 来做背负那些繁重的工作而不至于拖慢 Nagios 的运行。但是要注意，ePN 并不能加速插件本身(还要除去内嵌 Perl 解释器的加载时间)。如果要加速插件本身的执行，可以考虑使用 Perl XSUB 包(XS)或是 C，这么做的**前提是你已经确信插件 Perl 程序是足够优化并且保存了合理的算法**(用那个 Benchmark.pm 包来比较 Perl 程序模块的性能的方法是**没有意义的**)；
- 用 ePN 是一个学习 Perl 语言最好的机会。

12.1.3. 缺点

使用 ePN(Nagios 嵌入式 Perl 解释器)比之纯粹 Nagios 程序的缺点相近的,是在运行 Apache 带有 mod_perl (Apache 也是使用嵌入式解释器)和运行纯粹 Apache 程序的两种情形对比,情况也是这样:

- 一个 Perl 程序可能在使用纯粹 Nagios 程序时**运转良好**但可能在使用 ePN 时却可能**不正常**,要修改 Perl 插件程序以使之运转;
- 在使用 ePN 时调试 Perl 插件要比使用纯粹 Nagios 调试插件要困难一些;
- 使用 ePN 的插件比纯粹 Nagios 情况下要更大一些(内存占用);
- 想用的一些 Perl 结构体可能不能用或是用起来很困难;
- 不得不要关注'有多个进程在使用'和选择一种方式看起来更少交换或更少明显交互(注意用解释器执行是并发的——译者注);
- 要有更多的 Perl 功底(但不需要过多 Perl 技巧或素材——除非是使用 XSUBS 的插件)。

12.1.4. 使用嵌入式 Perl 解释器

如果要使用嵌入式 Perl 解释器来运行 Perl 插件和 Perl 脚本,下面这些是需要做的:

- 带有嵌入式 Perl 解释器支持选项来编译 Nagios(见下面的指令);
- 打开主配置文件里的 [enable_embedded_perl](#)选项开关;
- 设置 [use_embedded_perl_implicitly](#)选项以符合要求。该选项决定默认情况下是否要使用Perl解释器来运行个别Perl插件和脚本;
- 偶尔要对某些 Perl 插件或脚本要设置或打开或关闭使用嵌入式 Perl 解释器。这对于部分因带有嵌入式 Perl 解释的 Nagios 在执行 Perl 插件时存在问题时很有用,见下面指令是如何完成的。

12.1.5. 编译一个 Nagios 带嵌入式 Perl 解释器

如果要用嵌入式 Perl 解释器,首先需要编译 Nagios 支持它。只需要运行源程序配置脚本时带上 `--enable-embedded-perl` 参数即可。如果要嵌入式解释器缓存编译后的脚本,还要带上 `--with-perlcache` 参数,例子:

```
./configure --enable-embedded-perl --with-perlcache otheroptions...
```

一旦用新选项重新运行配置脚本,一定要重编译 Nagios。

12.1.6. 指定插件使用 Perl 解释器

自 Nagios 的第三版开始,可以在 Perl 插件或脚本中指定是否需要在嵌入式 Perl 解释器里运行。这对于运行那些在 Perl 解释器里运行有问题的 Perl 脚本的处理将非常有帮助。

明确地指定 Nagios 是否要使用嵌入式 Perl 解释器来运行某个 Perl 脚本,把下面的东西加到你的 Perl 插件或脚本里...

如果需要 Nagios 使用嵌入式 Perl 解释器来运行 Perl 脚本,在 Perl 程序里加入下面一行:

```
# nagios: +epn
```

如果不用嵌入式解释器，在 Perl 程序里加入下面这一行：

```
# nagios: -epn
```

上面的行必须出现在 Perl 脚本的前 10 行里以让 Nagios 检测 Perl 程序是否要用解释器。

提示



如果没有明确地用上述方式指出该插件是否要在 Perl 解释器里运行，Nagios 将自主决定。它取决于 [use_embedded_perl_implicitly](#) 变量设置。如果该变量值是 1，全部的 Perl 插件与脚本（那些没明确指定的）将在 Perl 解释器里运行，如果值是 0，那些没明确指定运行方式的插件和脚本将不会在解释器里运行。

12.1.7. 开发嵌入式 Perl 解释器可运行的 Perl 插件

更多有关开发嵌入式 Perl 解释器里可运行 Perl 插件的信息可查阅 [这篇文档](#)。

12.2. 开发使用内嵌式 Perl 解释器的 Nagios 插件

12.2.1. 介绍

Stanley Hopcroft 在嵌入式 Perl 解释器的工作机制方面卓有成效并且给出了利用嵌入式 Perl 解释器方面的优点和缺点的注释，他同时还给出了如何让 Perl 插件更好地在 Perl 解释器里运行的一些提示与帮助，下面内容多来自于他的注解。

需要注意本文档里用的“ePN”，它指定是带有嵌入式 Perl 解释器的 Nagios 版本，如果不是，Nagios 需要重新编译以带有嵌入式 Perl 解释器的支持。

12.2.2. 目标受众

- 中级 Perl 开发者—那些受 Perl 语言强大能力的熏陶却不会利用内在语言特性或是对那些特性掌握不多的开发员；
- 实用主义者而不是深入研究 Perl 语言特性的；
- 如果对 Perl 对象、命名空间管理、数据结构和调试等并不很感兴趣，或许也就够了；

12.2.3. 开发 Perl 插件必备知识

不管是否用 ePN 与否，下面的内容要注意：

- 总是要生成一些输出内容；
- 加上引用‘use utils’并引用些通用模块来输出（\$TIMEOUT %ERRORS &print_revision &支持等）；
- 总是知道一些 Perl 插件的标准习惯，如：
 - 退出时总是 exit 带着 \$ERRORS{CRITICAL}、\$ERRORS{OK} 等；
 - 使用 getopt 函数来处理命令行；

- 程序处理超时问题;
- 当没有命令参数时要给出可调用 print_usage;
- 使用标准的命令行选项开关(象-H 'host'、-V 'version'等)。

12.2.4. 开发 ePN 下的 Perl 插件必做内容

- 1、<DATA>不能使用，用下面方式替换：

```
• my $data = <<DATA;
• portmapper 100000
• portmap 100000
• sunrpc 100000
• rpcbind 100000
• rstatd 100001
• rstat 100001
• rup 100001
• ..
• DATA
•

%prognum = map { my($a, $b) = split; ($a, $b) } split(/\n/, $data) ;
```

- 2、BEGIN 块并不会象你想像的执行，尽可能避免用它;
- 3、确保编译时非常干净，象
 - use strict
 - use perl -w (其他开关特别是[-T]将不能用)
 - use perl -c
- 4、避免使用全程块里的变量语句(my)来传递到子程序里作为变量数据。如果子程序在插件时被多次调用这将是致命性错误。这样的子程序块起到一个“终止”作用，使得全程块里的变量被赋予第一个值而带入到后序子程序调用过程中导致死锁。但如果全程变量是只读的(比如是个可并发的结构体)就不是问题，在Bekman的[“推荐替代方式”](#)里是这样来做的：
 - 让子程序调用没有副作用方式的参考：

表 12.1. 源程序转换比对

转换前	转换后
-----	-----

转换前	转换后
<pre> my \$x = 1 ; sub a { .. Process \$x ... } . . a ; \$x = 2 ; a ; # anon closures __always__ rebind the current lexical value </pre>	<pre> my \$x = 1 ; \$a_cr = sub { ... Process \$x ... } ; . . &\$a_cr ; \$x = 2 ; &\$a_cr ; </pre>

- 把全程语句变量送到子程序块里(象个对象或模块那样来处理)；
- 将信息以引用或别名方式送入子程序里(\\$lex_var 或\$_[n])；
- 用包内全程替换语句并用‘use vars qw(global1 global2 ..)’从‘use strict’分离出来。
- 5、注意哪里可以得到更多信息。一般信息是要值得怀疑的(0'Reilly出版的书籍，加上Damien Conways著的“Object Oriented Perl”)，但真正有用的是从Stas Bekman在<http://perl.apache.org/guide/>里所写的mod_perl指南，这才是好的起点。那本书非常适用，虽然它里面没有一点关于Nagios的东西，全是如果来写在Apache里怎么开发利用嵌入式Perl解释器来写Perl程序的内容。提倡多使用perlembed的在线手册(manpage)。其基础是知道Lincoln Stein和Doug MacEachern写的有关Perl及嵌入Perl的一些东西，他们所著的‘Writing Apache Modules with Perl and C’一书绝对值得一读。
- 6、注意在 ePN 里插件有可能返回一些奇怪的值，这很有可能是由于前面 4 里所提到的问题所导致的；
 - 7、调试前做点准备：
 - 对 ePN 做个检查；
 - 插件里加些打印语句输出到标准错误设备 STDERR(不能用标准输出设备 STDOUT)来显示变量值；
 - 在 p1.p1 里加些打印语句来显示 ePN 有要执行插件时是如何看待插件的；
 - 在前台模式里运行 ePN(可能要结合前面的一些建议)；

- 插件里用'Deparse'模块来看一下解析命令选项在嵌入解释器实际得到的内容(见 Sean M. Burke 所写的'Constants in Perl', 在 Perl Journal 里, 写于 2001 年);

```
perl -MO::Deparse <your_program>
```

- 8、还要当心 ePN 把插件转换成什么了, 并且如果还有错误可以调试一下转换后的内容。正如下面看到的, p1.pl 把插件当作一个子程序调用, 子程序是在一个名为 'Embed::<something_related_to_your_plugin_file_name>' 的包里, 名字是'hndlr'。插件可能需要命令参数放在@ARGV 里, 因而 p1.pl 会把@_赋予@ARGV。这会按次序从'eval'取出值并且如果 eval 抛出一个错误(任意一个解析错误或运行错误)都会使插件中断运行。下面打印出了在 ePN 尝试运行之前由它转换好的 **check_rpc** 插件的真正内容, 大部分来自于插件的代码没有给出, 感兴趣的只是由 ePN 对插件的转换结果。为更清楚起见, 转换出的部分用下划线标识了一下:

```

•          package main;

•          use subs 'CORE::GLOBAL::exit';

•          sub CORE::GLOBAL::exit { die "ExitTrap: $_[0]

•      (Embed::check_5frpc)"; }

•          package Embed::check_5frpc; sub hndlr { shift(@_);

•      @ARGV=@_;

•      #! /usr/bin/perl -w

•      #

•      # check_rpc plugin for Nagios

•      #

•      # usage:

•      #      check_rpc host service

•      #

•      # Check if an rpc serice is registered and running

•      # using rpcinfo - $proto $host $prognum 2>&1 |";

•      #

•      # Use these hosts.cfg entries as examples

•      #

•      # command[check_nfs]=/some/path/libexec/check_rpc $HOSTADDRESS$ nfs

•      # service[check_nfs]=NFS;24x7;3;5;5;unix-admin;60;24x7;1;1;1;;check_rpc

```



```

• #
• # initial version: 3 May 2000 by Truongchinh Nguyen and Karl DeBisschop
• # current status: $Revision: 1.17 $
• #
• # Copyright Notice: GPL
• #
• ...
• rest of plugin code goes here (it was removed for brevity)
• ...
}

```

- 9、在插件里不用要‘use diagnostics’来运行 ePN 方式的程序，这会使得全部插件都返回一个“紧急”状态；
- 10、考虑用一个最小的 C 嵌入 Perl 的程序来检测插件。虽然它还不能担保在它里面运行正常的话也会在 ePN 里运行也能正常，但它可以找到很多 ePN 里肯定也会有的错误。[一个小的 ePN 程序已经在 Nagios 的源程序包里，放在 **contrib/**目录下，可用于测试 ePN 式的 Perl 插件，切换目录到 contrib/下并敲入‘make mini_epn’来编译它。它必须与 p1.pl 程序放在同一个目录里运行，p1.pl 也放进了 Nagios 源程序包里了。]

12.3. Nagios 插件 API

12.3.1. 其他资源

如果想给 Nagios 增加一个自己的插件，请访问如下资源：

- [Nagios插件项目官方网站](#)
- [Nagios插件开发的官方指南](#)

12.3.2. 插件概览

作为 Nagios 插件的脚本或执行程序必须(至少)要做两件事：

- 退出时给出几种可能的返回值中的一个；
- 至少要给出一条输出内容到标准输出设备(STDOUT)。

对 Nagios 来说，插件里面做什么并不重要。自制插件可以是做 TCP 端口状态检测，运行某个数据库查询，检查磁盘空闲空间，或其他需要检测的内容。这取决于你想检测什么东西，这完全由你自己决定。

12.3.3. 返回值

Nagios 用插件的返回值来生成主机或服务状态。下表里列出了合法的返回值以及对应的服务或主机状态。

表 12.2.

插件返回值	服务状态	主机状态
0	正常 (OK)	运行 (UP)
1	告警 (WARNING)	运行 (UP) 或宕机 (DOWN) / 不可达 (UNREACHABLE)*
2	紧急 (CRITICAL)	宕机 (DOWN) / 不可达 (UNREACHABLE)
3	未知 (UNKNOWN)	宕机 (DOWN) / 不可达 (UNREACHABLE)

注意



如果使能 [use aggressive host checking](#) 选项, 返回值 1 将使主机状态要么是宕机 (DOWN) 要么是不可达 (UNREACHABLE)。其他情况下, 返回值 1 将使主机状态是运行 (UP)。Nagios 将判定是宕机 (DOWN) 还是不可达 (UNREACHABLE) 状态的讨论在 [这篇文档](#) 里有。

12.3.4. 特定插件输出

最小情况下, 插件要返回一行文本输出。自 Nagios 3 版本起, 插件可以返回多行输出文本。插件可以返回性能数据以让外部应用来做后序处理。输出文本的基本格式如下:

```
TEXT OUTPUT | OPTIONAL PERFDATA

LONG TEXT LINE 1
LONG TEXT LINE 2
...
LONG TEXT LINE N | PERFDATA LINE 2
PERFDATA LINE 3
...
PERFDATA LINE N
```

性能数据 (用下划线示意的部分) 是可选的, 如果插件输出文本里有性能数据, 必须用管道符 (|) 把性能数据与其他数据分开, 额外的大段输出行 (用文字删除符示意的部分) 同样也是可选的。

12.3.5. 插件输出样例

下面看一下插件输出的样例...

案例 1: 只有一行文本输出 (不带性能数据)

假定插件的输出文本是这样：

DISK OK - free space: / 3326 MB (56%);

如果插件执行的是一个服务检测，整行输出都会保存在 [\\$SERVICEOUTPUT\\$](#) 宏里。

案例 2：一行输出带性能数据

插件的输出文本中带有性能数据可给外部应用来处理。性能数据要用管道符(|)分隔开，象是这样：

DISK OK - free space: / 3326 MB (56%); / /=2643MB;5948;5958;0;5968

如果插件执行的是一个服务检测，分隔符左侧的部分将保存在 [\\$SERVICEOUTPUT\\$](#) 宏里并且右侧(用下划线示意)的部分将保存在 [\\$SERVICEPERFDATA\\$](#) 宏里面。

案例 3：多行输出(正文和性能数据都有)

插件可以输出多行文本，并且带有正文输出和性能数据，象是这样：

DISK OK - free space: / 3326 MB (56%); / /=2643MB;5948;5958;0;5968

/ 15272 MB (77%);

/boot 68 MB (69%);

/home 69357 MB (27%);

/var/log 819 MB (84%); / /boot=68MB;88;93;0;98

/home=69357MB;253404;253409;0;253414

/var/log=818MB;970;975;0;980

如果插件执行的是一个服务检测，第一行分隔符左侧的部分将保存在 [\\$SERVICEOUTPUT\\$](#) 宏里，带有下划线标识的部分(带空格)将保存在 [\\$SERVICEPERFDATA\\$](#) 宏里，带删除符标识的部分(不带换行符)的部分将保存在 [\\$LONGSERVICEOUTPUT\\$](#) 宏里(以上的下划线和删除符只是为标记文本段而用的，实际文本中不带有符号格式——译者注)。

每个宏的最终结果是这样的：

表 12.3.

宏	内容
\$SERVICEOUTPUT\$	DISK OK - free space: / 3326 MB (56%);
\$SERVICEPERFDATA\$	/ /=2643MB;5948;5958;0;5968. /boot=68MB;88;93;0;98. /home=69357MB;253404;253409;0;253414. /var/log=818MB;970;975;0;980
\$LONGSERVICEOUTPUT\$	/ 15272 MB (77%); \n/boot 68 MB (69%); \n/var/log 819 MB (84%);

宏	内容
OUTPUT\$	

利用多行输出结果的机制，可以采取多种方式来返回性能数据：

- 无论什么情况都没有性能数据；
- 只返回一行性能数据；
- 只是在后序的行内返回性能数据(第一行不用的管道分隔符右侧不填内容)；
- 利用全部的输出位置来带出性能数据。

(看起来第一行右侧部分有点“多余”，真的可以不用，但其实这是作者为软件向下兼容低版本使用的插件而特意这么做的，很有必要这么做，看一下源程序就明白了。——译者注)

12.3.6. 插件输出长度限制

Nagios 只处理插件返回的前 4KB 数据内容。这样是为了防止插件返回一个上兆或上千兆的数据块给 Nagios 处理。这个 4K 的限制很容易修改，如果你想改，可以编辑一下源代码里的 MAX_PLUGIN_OUTPUT_LENGTH 宏定义，在源程序包的 `include/nagios.h.in` 文件里，重编译一下 Nagios 就可以了，其他地方不用动！

12.3.7. 例子

如果想找点例子来学习开发插件，推荐去下载Nagios插件项目官方的软件包，插件代码使用多种语言(象C、Perl和SHELL脚本等)写成插件。取得Nagios插件的官方代码包的方法可以看 [这篇文档](#)。

12.3.8. Perl 插件

Nagios可以选择支持 [嵌入式Perl解释器](#)可以提高执行Perl插件的速度。更多有关使用嵌入式Perl解释器来开发Perl插件的信息可以查阅 [这篇文档](#)。