# Mt-Falcon——Open-Falcon在美团点评的应用与实践 –

http://tech.meituan.com/Mt-Falcon\_Monitoring\_System.html

## 前言

监控系统是整个业务系统中至关重要的一环，它就像眼睛一样，时刻监测机房、网络、服务器、应用等运行情况，并且在出现问题时能够及时做出相应处理。

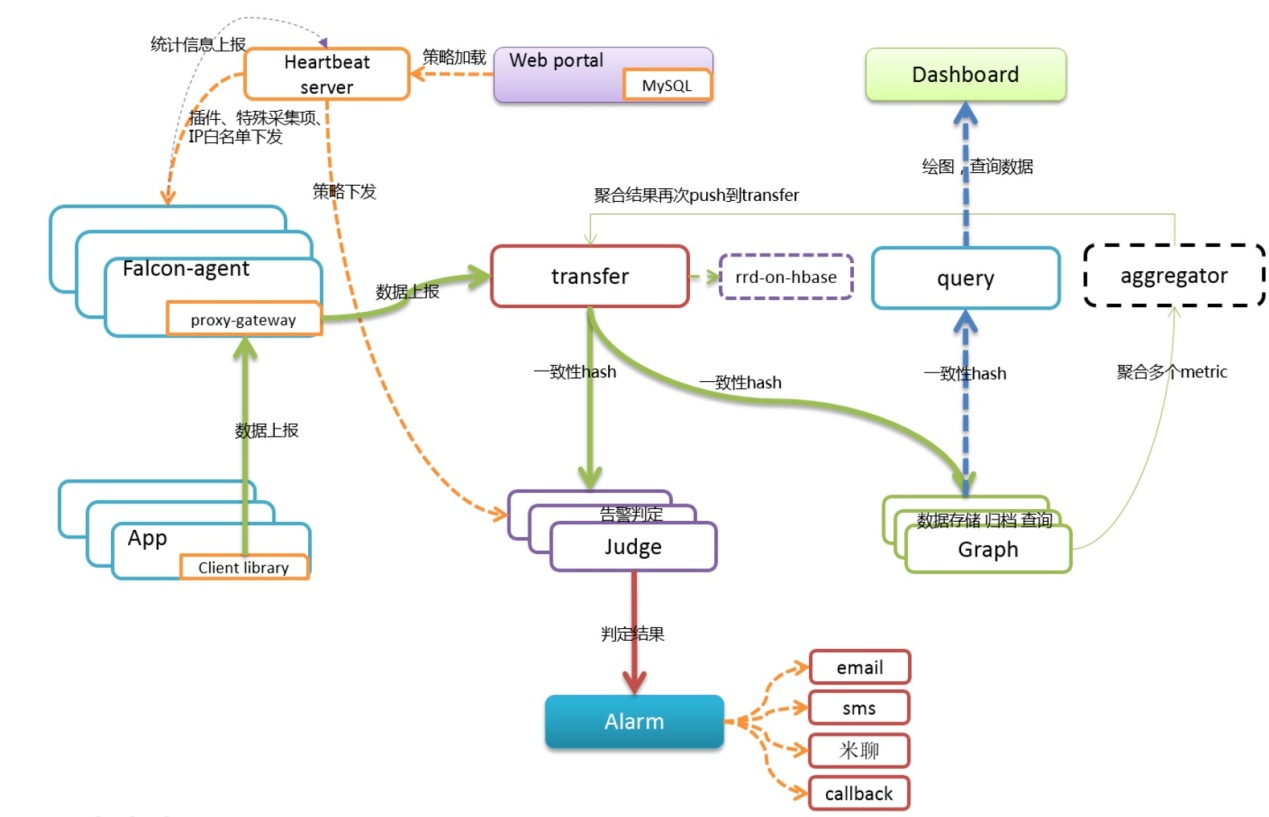
美团点评刚开始使用的是Zabbix监控系统，几经优化，在当时能够达到2W+机器，450W+监控项的量。随着各业务线的发展，监控项越来越多，Zabbix的问题也越来越突出，当时针对Zabbix的吐槽问题有：

* 不支持扩展，本身是一个单点，当机器规模超过万台的时候会出现很明显的性能问题。
* 改造难度比较大，不支持定制化功能。
* 配置比较复杂，学习成本较高。
* 对外提供的API不够丰富，很难与其他业务系统集成。

这个时候我们急于寻找一个替代的解决方案，经过筛选后，最终选择引进最初由小米开源的[Open-Falcon监控系统](https://github.com/open-falcon)（[文档](http://book.open-falcon.org/zh/index.html)）。

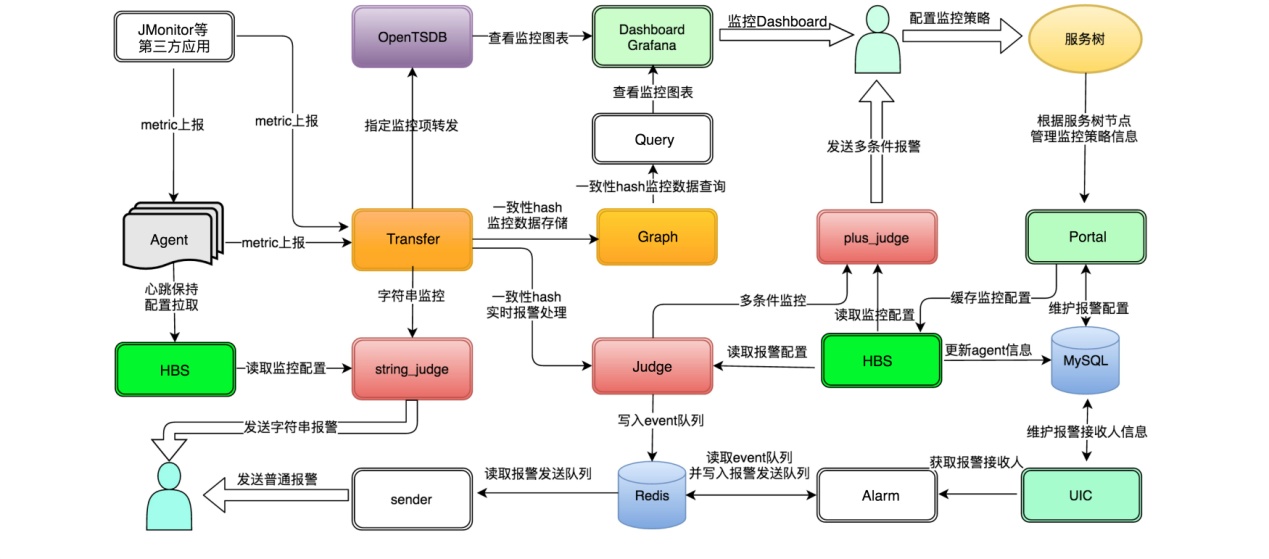
下面本文将为大家详细介绍Mt-Falcon在原来Open-Falcon的基础上做出的一些改进。

Open-Falcon架构图



图片转载自[Open-Falcon官网](https://book.open-falcon.org/zh/intro/index.html)

Mt-Falcon的架构图



Mt-Falcon相对Open-Falcon改造后，比较大的功能点有：报警禁用、报警ACK、报警升级、报警任务分布式消费、支持OpenTSDB存储、字符串监控、多条件监控、索引信息存储改造、过期索引信息自动删除且重新上报后会再次重建等。

改进列表

## 一、Agent改造

1. 提升Agent数据转发的性能

异步化处理。之前是每调一次Agent的上报接口就会往Transfer上报一次，在数据量特别大，每次发送条数又比较少的情况下，有可能会出现数据丢失的情况。现在会把监控数据先缓存在Agent程序缓存里面，每0.5秒往Transfer上报一次，每次最多发送1W条监控项。只要一个上报周期（默认60s）上报的监控项个数<100W就不会出现性能问题。

2. 上报网卡流量时标识出机器不同的网卡类型

业务方的机器有可能一部分在千兆集群上，一部分在万兆集群上，不是很好区分。之前配置网卡监控的时候统一应用的是千兆网卡的监控指标，这样就造成了万兆集群上面机器的网卡误报警。在Agent采集网卡指标的时候自动打上网卡类型的Tag，就解决了上面的问题。现在机器网卡类型主要有4种，千兆、万兆、双千兆、双万兆，配置监控策略时可以根据不同的网卡类型设置不同的报警阈值。

3. 支持进程级别的coredump监控

这个类似于普通的进程监控，当检测到指定进程出现core时，上报特定的监控指标，根据这个监控指标配置相应的报警策略即可。

4. 日志自动切分

正常情况下Agent的日志量是很小的，对，正常情况下。但是，凡事总有意外，线上出现过Falcon-Agent的日志量达到过100多G的情况。

为了解决这个问题，我们引入了一个新的日志处理库Go-Logger。Go-Logger库基于对Golang内置Log的封装，可以实现按照日志文件大小或日志文件日期的方式自动切分日志，并支持设置最大日志文件保存份数。改进版的Go-Logger，只需要把引入的Log包替换成Go-Logger，其他代码基本不用变。

Go-Logger的详细用法可参考：<https://github.com/donnie4w/go-logger>。

5. 解决机器hostname重复的问题

系统监控项指标上报的时候会自动获取本地的hostname作为Endpoint。一些误操作，如本来想执行host命令的，一不小心执行了hostname，这样的话本地的hostname就被人为修改了，再上报监控项的时候就会以新的hostname为准，这样就会导致两台机器以相同hostname上报监控项，造成了监控的误报。

为了解决这一问题，Falcon-Agent获取hostname的方式改为从/etc/sysconfig/network文件中读取，这样就避过了大部分的坑。另外，当已经发生这个问题的时候怎么快速定位到是哪台机器hostname出错了呢？这个时候可以选择把机器的IP信息作为一个监控指标上报上来。

6. 支持Falcon-Agent存活监控

Falcon-Agent会与HBS服务保持心跳连接，利用这个特性来监控Falcon-Agent实例的存活情况，每次心跳连接都去更新Redis中当前Falcon-Agent对应的心跳时间戳。

另外，启动一个脚本定时获取Redis中所有的Falcon-Agent对应的时间戳信息，并与当前时间对应的时间戳做比对，如果当前时间对应的时间戳与Falcon-Agent的时间戳的差值大于5分钟，则认为该Falcon-Agent跪掉了，然后触发一系列告警。

## 二、HBS改造

1. 内存优化

在进行数据通信的时候有两点比较重要，一个是传输协议，一个是数据在传输过程中的编码协议。

HBS（HeartBeat Server）和Judge之间的通信，之前是使用JSON-RPC框架进行的数据传输。JSON-RPC框架使用的传输协议是RPC（RPC底层其实是TCP），编码协议使用的是Go自带的encoding/json。

由于encoding/json在进行数据序列化和反序列化时是使用反射实现的，导致执行效率特别低，占用内存也特别大。线上我们HBS实例占用的最大内存甚至达到了50多G。现在使用RPC+MessagePack代替JSON-RPC，主要是编码协议发生了变化，encoding/json替换成了MessagePack。

MessagePack是一个高效的二进制序列化协议，比encoding/json执行效率更高，占用内存更小，优化后HBS实例最大占用内存不超过6G。

关于RPC和MessagePack的集成方法可以参考：<https://github.com/ugorji/go/tree/master/codec#readme>。

2. 提供接口查询指定机器对应的聚合后的监控策略列表

机器和Group关联，Group和模板关联，模板与策略关联，模板本身又支持继承和覆盖，所以最终某台机器到底对应哪些监控策略，这个是很难直观看到的。但这个信息在排查问题的时候又很重要，基于以上考虑，HBS开发了这么一个接口，可以根据HostID查询当前机器最终应用了哪些监控策略。

3. 解决模板继承问题，现在继承自同一个父模板的两个子模板应用到同一个节点时只有一个子模板会生效

两个子模板继承自同一个父模板，这两个子模板应用到同一个节点时，从父模板中继承过来的策略只会有一个生效，因为HBS在聚合的时候会根据策略ID去重。如果两个子模板配置的是不同的报警接收人，则有一个模板的报警接收人是收不到报警的。

为了解决这个问题，改为HBS在聚合的时候根据策略ID+ActionID去重，保证两个子模板都会生效。

4. 报警禁用

对于未来可以预知的事情，如服务器重启、业务升级、服务重启等，这些都是已知情况，报警是可以暂时禁用掉的。

为了支持这个功能，我们现在提供了5种类型的报警禁用类型：

* 机器禁用：会使这台机器的所有报警都失效，一般在机器处于维修状态时使用。
* 模板禁用：会使模板中策略全部失效，应用此模板的节点都会受到影响。
* 策略禁用：会使当前禁用的策略失效，应用此策略对应模板的节点都会受到影响。
* 指定机器下的指定策略禁用：当只想禁用指定机器下某个策略时可以使用此方式，机器的其他监控策略不受影响。
* 指定节点下的指定模板禁用：这个功能类似于解除该模板与节点的绑定关系，唯一不同点是禁用后会自动恢复。

为了避免执行完禁用操作后，忘记执行恢复操作，造成监控一直处于禁用状态，我们强制不允许永久禁用。

目前支持的禁用时长分别为，禁用10分钟、30分钟、1小时、3小时、6小时、1天、3天、一周、两周。

三、Transfer改造

1. Endpoint黑名单功能

Falcon的数据上报方式设计的特别友好，在很大程度上方便了用户接入，不过有时也会带来一些问题。有业务方上报数据的时候会把一些变量（如时间戳）作为监控项的构成上报上来，因为Transfer端基本没有做数据的合法性校验，这样就造成了某个Endpoint下面对应大量的监控项，曾经出现过一个Endpoint下面对应数千万个监控项。这对索引数据数据的存储和查询性能都会有很大的影响。

为了解决这个问题，我们在Transfer模块开发了Endpoint黑名单功能，支持禁用整个Endpoint或者禁用Endpoint下以xxx开头的监控指标。再出现类似问题，可与业务方确认后立即禁用指定Endpoint数据的上报，而不会影响其他正常的数据上报。

2. 指定监控项发送到OpenTSDB

有些比较重要的监控指标，业务方要求可以看到一段时间内的原始数据，对于这类特殊的指标现在的解决方案是转发到OpenTSDB里面保存一份。Transfer会在启动时从Redis里面获取这类特定监控项，然后更新到Transfer自己的缓存中。当Redis中数据发生变更时会自动触发Transfer更新缓存，以此来保证数据的实时性和Transfer本身的性能。

四、Judge改造

1. 内存优化

有很多监控指标上报上来后，业务方可能只是想在出问题时看下监控图，并不想配置监控策略。据统计，80%的监控指标都是属于这种。之前Judge的策略是只要数据上报就会在Judge中缓存一份，每个监控指标缓存最近上报的11个数据点。

其实，对于没有配置监控策略的监控指标是没有必要在Judge中缓存的。我们针对这种情况做了改进，Judge只缓存配置监控策略的监控项数据，对于没有配置监控策略的监控项直接忽略掉。

2. 报警状态信息持久化到本地，解决Judge重启报警重复发出的问题

之前的报警事件信息都是缓存到Judge内存中的，包括事件的状态、事件发送次数等。Judge在重启的时候这些信息会丢掉，造成之前未恢复的报警重复发出。

现在改成Judge在关闭的时候会把内存中这部分信息持久化到本地磁盘（一般很小，也就几十K到几M左右），启动的时候再把这些信息Load进Judge内存，这样就不会造成未恢复报警重复发出了。

据了解，小米那边是通过改造Alarm模块实现的，通过比对报警次数来判断当前是否发送报警，也是一种很好的解决方案。

3. 报警升级

我们现在监控模板对应的Action里面有第一报警接收组和第二报警接收组的概念。当某一事件触发后默认发给第一报警接收组，如果该事件20分钟内没有解决，则会发给第二报警接收组，这就是报警升级的含义。

4. 报警ACK

ACK的功能跟Zabbix中的ACK是一致的，当已经确认了解到事件情况，不想再收到报警时，就可以把它ACK掉。

ACK功能的大致实现流程是：

* Alarm发送报警时会根据Endpoint+Metric+Tags生成一个ACK链接，这个链接会作为报警内容的一部分。
* 用户收到报警后，如果需要ACK掉报警，可以点击这个链接，会调用Transfer服务的ACK接口。
* Transfer收到ACK请求后，会根据传输过来的Endpoint+Metric+Tags信息把这个请求转发到对应的Judge实例上，调用Judge实例的ACK接口。
* Judge收到ACK请求后，会根据EventID把缓存中对应的事件状态置为已ACK，后续就不会再发送报警。

5. Tag反选

大家都知道配置监控策略时善用Tag，可以节省很多不必要的监控策略。比方说我想监控系统上所有磁盘的磁盘空间，其实只需要配置一条监控策略，Metric填上df.bytes.free.percent就可以，不用指定Tags，它就会对所有的磁盘生效。

这个时候如果想过滤掉某一块特殊的盘，比方说想把mount=/dev/shm这块盘过滤掉，利用Tag反选的功能也只需要配置一条监控策略就可以，Metric填上df.bytes.free.percent，Tags填上^mount=/dev/shm即可，Judge在判断告警的时候会自动过滤掉这块盘。

6. 多条件报警转发到plus\_judge

Judge在收到一个事件后，会首先判断当前事件是否属于多条件报警的事件，事件信息是在配置监控策略的时候定义的。如果属于多条件报警的事件，则直接转发给多条件报警处理模块plus\_judge。关于plus\_judge，后面会重点介绍。

五、Graph改造

1. 索引存储改造

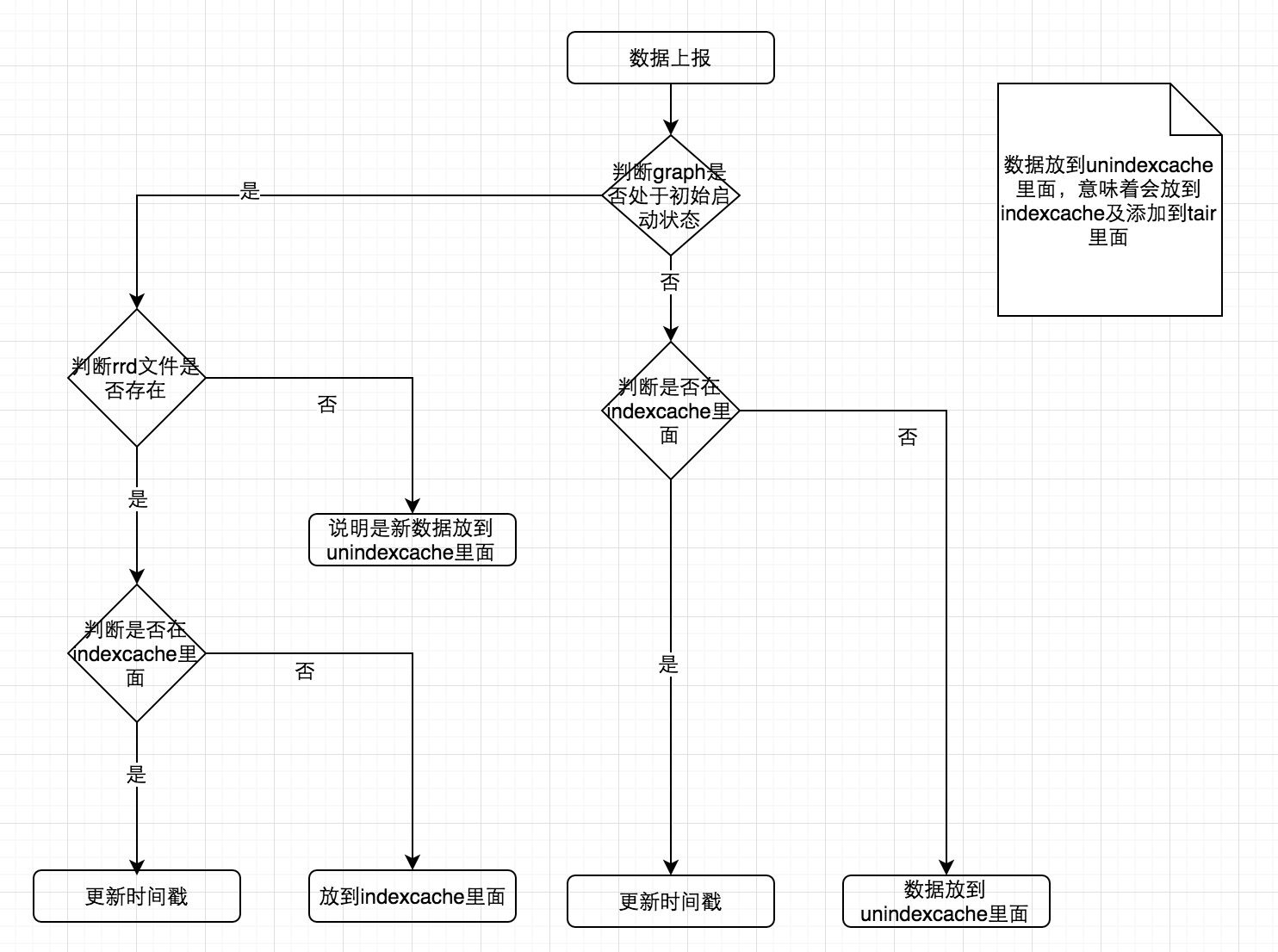
索引存储这块目前官方的存储方式是MySQL，监控项数量上来后，很容易出现性能问题。我们这边的存储方式也是改动了很多次，现在是使用Redis+Tair实现的。

建议使用Redis Cluster，现在Redis Cluster也有第三方的Go client了。详情请参考：<https://github.com/chasex/redis-go-cluster>。

官方我看也在致力于索引存储的改造，底层使用BoltDB存储，具体的可参考小米来炜的Git仓库：<https://github.com/laiwei/falcon-index>。

这块我们有专门做过Redis、Tair、BoltDB的性能测试，发现这三个存储在性能上差别不是很大。

2. 过期索引自动删除且重新上报后会自动重建

监控项索引信息如果超过1个月时间没有数据上报，则Graph会自动删除该索引，删除Tair中存储的索引信息时会同步删除indexcache中缓存的索引信息。  
索引删除后，如果对应数据又重新进行上报，则会重新创建对应的索引信息。  


默认Graph在刚启动的6个小时内（时间可配置）定为初始启动状态，数据放到unindexcache队列里面，意味着会重新创建索引。

3. 解决查询历史数据时最新的数据点丢失的问题

改造之前：

* 查询12小时内的监控数据时，会先从RRD文件取数据，再把取到的数据与缓存中的数据集成。集成原则是RRD与缓存中相同时间点的数据一律替换为缓存中的数据，所以查询12小时内的数据是可以正常返回的。
* 查询超过12小时内的数据时，会直接从RRD文件获取，不再与缓存中数据集成，所以在取超过12小时内的数据时，最新的数据上报点的数据一直是空的。

改造之后：

* 查询12小时内的数据，处理原则不变。
* 查询超过12小时内的数据时，先从RRD文件获取，再与缓存中数据集成。集成原则是RRD与缓存中相同时间点的数据，如果RRD数据为空，则替换为缓存中的数据，如果RRD数据不为空，则以RRD数据为准。

这里有一个问题，超过12小时内的数据都是聚合后的数据，缓存中的数据都是原始值，相同时间点RRD中为空的数据替换为缓存中的数据，相当于聚合后的数据用原始数据替换掉了，是有一定误差的，不过有胜于无。

六、Alarm改造

1. 报警合并

重写了Alarm模块的报警合并逻辑：

* 所有报警都纳入合并范畴
* 按照相同Metric进行合并
* 前3次直接发，后续每分钟合并一次
* 如果5分钟内没有报警，则下次重新计数

2. 报警发散

报警发散的作用是在宿主机特定监控指标触发后，不仅要发给配置的报警组，还要发给宿主机上虚拟机对应的负责人。

现在需要发散的宿主机特定监控指标有：

* net.if.in.Mbps 网卡入流量
* net.if.out.Mbps 网卡出流量
* icmp.ping.alive 机器存活监控
* cpu.steal CPU偷取

根据机器所属的环境不同，发给对应的负责人：

* prod环境：发给SRE负责人+RD负责人
* staging环境：发给RD责人
* test环境：发给测试负责人

3. 报警白名单

报警白名单是指当某一个指定的Metric发生大量告警时，可以迅速屏蔽掉这个Metric的告警，这个比较简单，就不多说了。

4. 报警任务分布式消费

未恢复报警信息之前是存储在Alarm内存里面，现在改为存储到Redis中。这样就可以启动多个Alarm实例，同时从Redis报警队列中取任务进行消费。Redis本身是单线程的，可以保证一个报警发送任务只会被一个Alarm实例获取到，sender模块使用同样逻辑处理，从而实现了报警任务的分布式消费处理。

5. 报警方式改造

现在报警方式和事件优先级解绑，优先级只是表示故障的严重程度，具体报警发送方式可以根据个人喜好自行选择。在美团点评内部，可以通过内部IM大象、邮件、短信和电话等方式发送报警。

现在支持的优先级有：

* p0: 最高优先级，强制发送短信，大象和邮件可以自行选择。
* p1: 高优先级，强制发送短信，大象和邮件可以自行选择。
* p2: 普通优先级，强制发送大象，邮件可以自行选择。
* p3: 低优先级，只发送邮件。
* p9: 特殊优先级，强制使用电话+短信+大象+邮件进行发送。

6. 报警持久化和报警统计

报警持久化这块刚开始使用的是InfluxDB，不过InfluxDB不方便统计，而且还有一些其它方面的坑，后来就改成直接存到MySQL中了。

我们每天会对报警信息做一个统计，会按服务、人、机器和监控项的维度分别给出Top10。

还会给出最近7天的报警量变化趋势，以及在每个BG内部分别按服务、机器、人的维度给出当前Top20的异常数和周同比。

7. 报警红盘

报警红盘的作用是统计一段时间内每个服务新触发报警的数量，把Top10的服务展示到页面上。报警数>=90显示红色，50~90之间显示黄色，当有事故发生时基本可以从红盘上观察出来哪个服务出现了事故，以及事故的影响范围。

8. 监控模板支持发给负责人的选项

监控模板对应的Action中添加一个发给负责人的选项，这样Action中的报警组可以设置为空，在触发报警的时候会自动把报警信息发给相应的负责人。可以实现不同机器的报警发给不同的接收人这个功能。

9. 触发base监控的时候自动发给相应负责人

为避免报警消息通知不到位，我们设置了一批基础监控项，当基础监控项触发报警的时候，会自动发给相应的负责人。

基础监控项有：

* "net.if.in.Mbps",
* "net.if.out.Mbps",
* "cpu.idle",
* "df.bytes.free.percent",
* "df.inodes.free.percent",
* "disk.io.util",
* "icmp.ping.alive",
* "icmp.ping.msec",
* "load.1minPerCPU",
* "mem.swapused.percent",
* "cpu.steal",
* "kernel.files.percent",
* "kernel.coredump",
* "df.mounts.ro",
* "net.if.change",

七、Portal/Dashboard改造

1. 绑定服务树

创建模板，添加策略，配置报警接收人等操作都是在服务树上完成。

2. 提供一系列接口，支持所有操作接口化，并对接口添加权限认证

我们支持通过调用API的方式，把监控功能集成到自己的管理平台上。

3. 记录操作日志

引入公司统一的日志处理中心，把操作日志都记录到上面，做到状态可追踪。

4. shift多选功能

在Dashboard查看监控数据时支持按住shift多选功能。

5. 绘图颜色调整

绘图时线条颜色统一调成深色的。

6. 索引自维护

系统运行过程中会出现部分索引的丢失和历史索引未及时清除等问题，我们在Dashboard上开放了一个入口，可以很方便地添加新的索引和删除过期的索引。

7. Dashboard刷新功能

通过筛选Endpoint和Metric查看监控图表时，有时需要查看最新的监控信息，点击浏览器刷新按钮在数据返回之前页面上会出现白板。为了解决这个问题我们添加了一个刷新按钮，点击刷新按钮会自动显示最近一小时内的监控数据，在最新的数据返回之前，原有页面不变。

8. screen中单图刷新功能

screen中图表太多的话，有时候个别图表没有刷出来，为了看到这个图表还要刷新整个页面，成本有点高。所以我们做了一个支持单个图表刷新的功能，只需要重新获取这单个图表的数据即可。

9. 支持按环境应用监控模板

现在支持把监控模板应用到指定的环境上，比方说把一个模板直接应用到某个业务层节点的prod环境上，这样只会对业务层节点或者业务层节点的子节点的prod环境生效，staging和test环境没有影响。

## 八、新增模块

1. Ping监控

使用Fping实现的Ping存话监控和延迟监控，每个机房有自己的Ping节点，不同节点之前互相Ping，实现跨机房Ping监控。

2. 字符串监控

字符串监控跟数值型监控共用一套监控配置，也就是Portal和HBS是统一的。当上报上来的数据是字符串类型，会交由专门的字符串处理模块string\_judge处理。

3. 同比环比监控

同比环比监控类似于nodata的处理方式，自行设定跟历史某个时间点数据做对比。因为数据会自动聚合，所以与历史上某个时间点做对比的话，是存在一定误差的。

官方提供了diff和pdiff函数，如果是对比最近10个数据点的话，可以考虑使用这种方式。也可以考虑把需要做同比环比监控的监控指标存入到OpenTSDB中，做对比的时候直接从OpenTSDB获取历史数据。

4. 多条件监控

有些异常情况，可能单个指标出现问题并没有什么影响，想实现多个指标同时触发的时候才发报警出来。因为Judge是分布式的，多个指标很可能会落到不同的Judge实例上，所以判断起来会比较麻烦。后来我们做了一个新的模块plus\_judge，专门用来处理多条件告警情况。

实现方案是：

* 组成多条件监控的多个策略，按照策略ID正序排序后，生成一个唯一序列号，这些策略在存储的时候会一并存下额处的3个信息，是否属于多条件监控，序列号，组成这个多条件监控的策略个数。
* Judge在收到有多条件告警标识的策略触发的告警事件时，直接转发给多条件监控处理模块plus\_judge。
* plus\_judge会根据序列号和多条件个数，判断是否多个条件都同时满足，如果全都满足，则才会发报警。

总结

Mt-Falcon现在在美团点评已经完全替换掉Zabbix监控，接入美团点评所有机器，数据上报QPS达到100W+，总的监控项个数超过两个亿。下一步工作重点会主要放在美团点评监控融合统一，配置页面改造，报警自动处理，数据运营等方面。

我们也一直致力于推动Open-Falcon社区的发展，上面所列部分Feature已Merge到官方版本，后面也会根据需求提相应PR过去。

作者简介

大闪，美团点评SRE组监控团队负责人。曾就职于高德、新浪，2015年加入原美团，一直负责监控体系建设。目前致力于故障自动追踪与定位、故障自动处理、数据运营等，持续提升监控系统稳定性、易用性和拓展性。

**不想错过技术博客更新？想给文章评论、和作者互动？第一时间获取技术沙龙信息？**

**请关注我们的官方微信公众号“美团点评技术团队”。现在就拿出手机，扫一扫：**

# 详解coredump - tenfyguo的技术专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/tenfyguo/article/details/8159176/

**一，什么是coredump**

        我们经常听到大家说到程序core掉了，需要定位解决，这里说的大部分是指对应程序由于各种异常或者bug导致在运行过程中异常退出或者中止，并且在满足一定条件下（这里为什么说需要满足一定的条件呢？下面会分析）会产生一个叫做core的文件。

        通常情况下，core文件会包含了程序运行时的内存，寄存器状态，堆栈指针，内存管理信息还有各种函数调用堆栈信息等，我们可以理解为是程序工作当前状态存储生成第一个文件，许多的程序出错的时候都会产生一个core文件，通过工具分析这个文件，我们可以定位到程序异常退出的时候对应的堆栈调用等信息，找出问题所在并进行及时解决。

**二，coredump文件的存储位置**

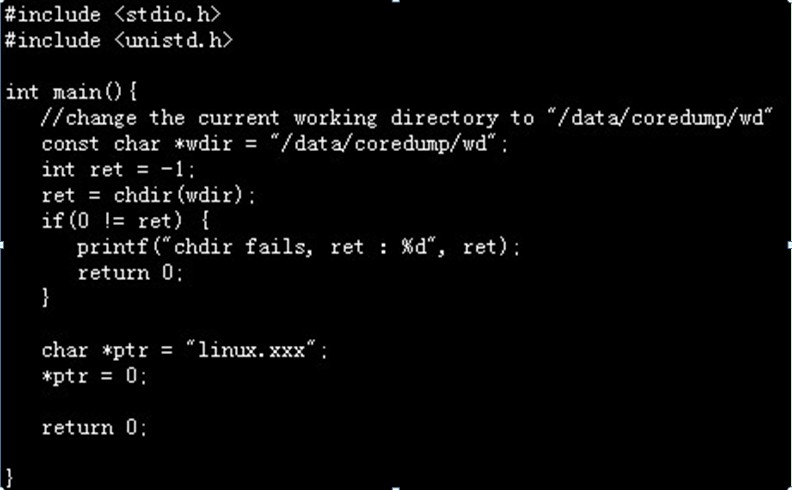
   core文件默认的存储位置与对应的可执行程序在同一目录下，文件名是core，大家可以通过下面的命令看到core文件的存在位置：

   cat  /proc/sys/kernel/core\_pattern

   缺省值是core

**注意：**这里是指在进程当前工作目录的下创建。通常与程序在相同的路径下。但如果程序中调用了chdir函数，则有可能改变了当前工作目录。这时core文件创建在chdir指定的路径下。有好多程序崩溃了，我们却找不到core文件放在什么位置。和chdir函数就有关系。当然程序崩溃了不一定都产生 core文件。

如下程序代码：则会把生成的core文件存储在/data/coredump/wd，而不是大家认为的跟可执行文件在同一目录。



通过下面的命令可以更改coredump文件的存储位置，若你希望把core文件生成到/data/coredump/core目录下：

   echo “/data/coredump/core”> /proc/sys/kernel/core\_pattern

注意，这里当前用户必须具有对/proc/sys/kernel/core\_pattern的写权限。

缺省情况下，内核在coredump时所产生的core文件放在与该程序相同的目录中，并且文件名固定为core。很显然，如果有多个程序产生core文件，或者同一个程序多次崩溃，就会重复覆盖同一个core文件，因此我们有必要对不同程序生成的core文件进行分别命名。

我们通过修改kernel的参数，可以指定内核所生成的coredump文件的文件名。例如，使用下面的命令使kernel生成名字为core.filename.pid格式的core dump文件：

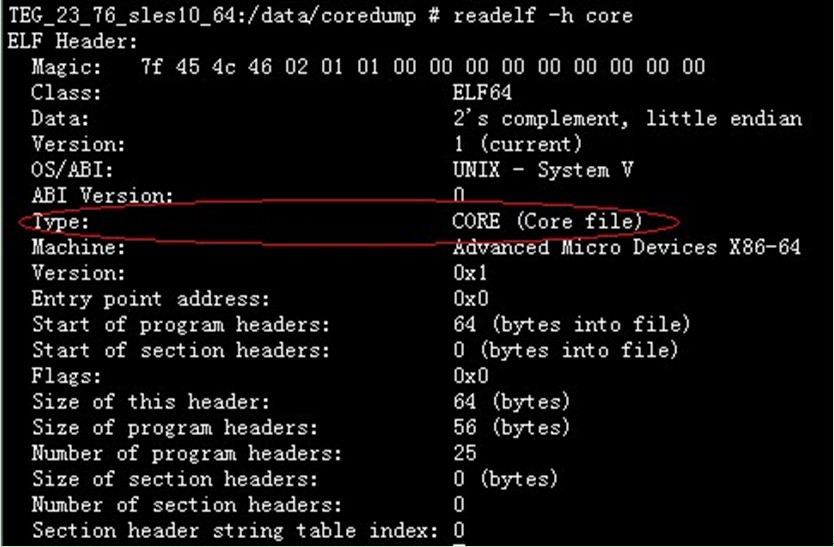
echo “/data/coredump/core.%e.%p” >/proc/sys/kernel/core\_pattern

这样配置后，产生的core文件中将带有崩溃的程序名、以及它的进程ID。上面的%e和%p会被替换成程序文件名以及进程ID。

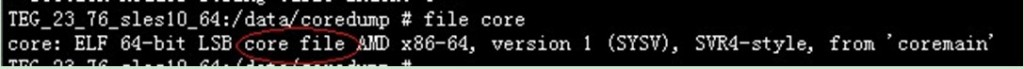
如果在上述文件名中包含目录分隔符“/”，那么所生成的core文件将会被放到指定的目录中。 需要说明的是，在内核中还有一个与coredump相关的设置，就是/proc/sys/kernel/core\_uses\_pid。如果这个文件的内容被配置成1，那么即使core\_pattern中没有设置%p，最后生成的core dump文件名仍会加上进程ID。

**三，如何判断一个文件是coredump文件？**

在类unix系统下，coredump文件本身主要的格式也是ELF格式，因此，我们可以通过readelf命令进行判断。



     可以看到ELF文件头的Type字段的类型是：CORE (Core file)

     可以通过简单的file命令进行快速判断：     

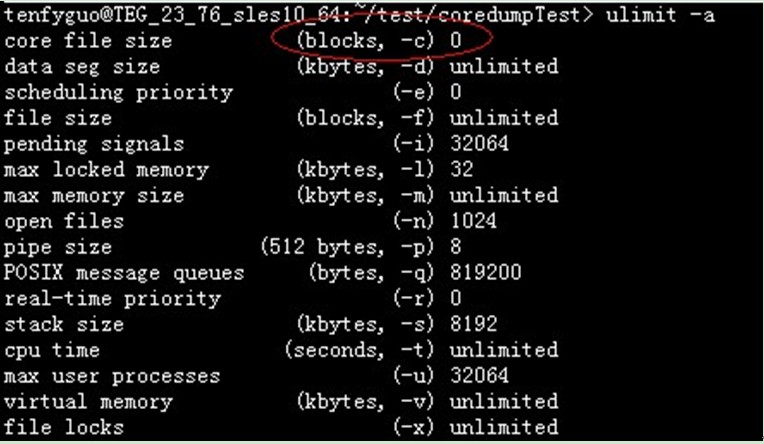
**四，产生coredum的一些条件总结**

1，  产生coredump的条件，首先需要确认**当前会话**的ulimit –c，若为0，则不会产生对应的coredump，需要进行修改和设置。

ulimit  -c unlimited  (可以产生coredump且不受大小限制)

若想甚至对应的字符大小，则可以指定：

ulimit –c [size]



       可以看出，这里的size的单位是blocks,一般1block=512bytes

        如：

        ulimit –c 4  (注意，这里的size如果太小，则可能不会产生对应的core文件，笔者设置过ulimit –c 1的时候，系统并不生成core文件，并尝试了1，2，3均无法产生core，至少需要4才生成core文件)

但当前设置的ulimit只对当前会话有效，若想系统均有效，则需要进行如下设置：

Ø  在/etc/profile中加入以下一行，这将允许生成coredump文件

ulimit-c unlimited

Ø  在rc.local中加入以下一行，这将使程序崩溃时生成的coredump文件位于/data/coredump/目录下:

echo /data/coredump/core.%e.%p> /proc/sys/kernel/core\_pattern

注意rc.local在不同的环境，存储的目录可能不同，susu下可能在/etc/rc.d/rc.local

      更多ulimit的命令使用，可以参考：<http://baike.baidu.com/view/4832100.htm>

      这些需要有root权限, 在ubuntu下每次重新打开中断都需要重新输入上面的ulimit命令, 来设置core大小为无限.

2， 当前用户，即执行对应程序的用户具有对写入core目录的写权限以及有足够的空间。

3， 几种不会产生core文件的情况说明：

The core file will not be generated if

(a)    the process was set-user-ID and the current user is not the owner of the program file, or

(b)     the process was set-group-ID and the current user is not the group owner of the file,

(c)     the user does not have permission to write in the current working directory,

(d)     the file already exists and the user does not have permission to write to it, or

(e)     the file is too big (recall the RLIMIT\_CORE limit in Section 7.11). The permissions of the core file (assuming that the file doesn't already exist) are usually user-read and user-write, although Mac OS X sets only user-read.

**五，coredump产生的几种可能情况**

造成程序coredump的原因有很多，这里总结一些比较常用的经验吧：

 1，内存访问越界

  a) 由于使用错误的下标，导致数组访问越界。

  b) 搜索字符串时，依靠字符串结束符来判断字符串是否结束，但是字符串没有正常的使用结束符。

  c) 使用strcpy, strcat, sprintf, strcmp,strcasecmp等字符串操作函数，将目标字符串读/写爆。应该使用strncpy, strlcpy, strncat, strlcat, snprintf, strncmp, strncasecmp等函数防止读写越界。

 2，多线程程序使用了线程不安全的函数。

应该使用下面这些可重入的函数，它们很容易被用错：

asctime\_r(3c) gethostbyname\_r(3n) getservbyname\_r(3n)ctermid\_r(3s) gethostent\_r(3n) getservbyport\_r(3n) ctime\_r(3c) getlogin\_r(3c)getservent\_r(3n) fgetgrent\_r(3c) getnetbyaddr\_r(3n) getspent\_r(3c)fgetpwent\_r(3c) getnetbyname\_r(3n) getspnam\_r(3c) fgetspent\_r(3c)getnetent\_r(3n) gmtime\_r(3c) gamma\_r(3m) getnetgrent\_r(3n) lgamma\_r(3m) getauclassent\_r(3)getprotobyname\_r(3n) localtime\_r(3c) getauclassnam\_r(3) etprotobynumber\_r(3n)nis\_sperror\_r(3n) getauevent\_r(3) getprotoent\_r(3n) rand\_r(3c) getauevnam\_r(3)getpwent\_r(3c) readdir\_r(3c) getauevnum\_r(3) getpwnam\_r(3c) strtok\_r(3c) getgrent\_r(3c)getpwuid\_r(3c) tmpnam\_r(3s) getgrgid\_r(3c) getrpcbyname\_r(3n) ttyname\_r(3c)getgrnam\_r(3c) getrpcbynumber\_r(3n) gethostbyaddr\_r(3n) getrpcent\_r(3n)

 3，多线程读写的数据未加锁保护。

对于会被多个线程同时访问的全局数据，应该注意加锁保护，否则很容易造成coredump

 4，非法指针

  a) 使用空指针

  b) 随意使用指针转换。一个指向一段内存的指针，除非确定这段内存原先就分配为某种结构或类型，或者这种结构或类型的数组，否则不要将它转换为这种结构或类型的指针，而应该将这段内存拷贝到一个这种结构或类型中，再访问这个结构或类型。这是因为如果这段内存的开始地址不是按照这种结构或类型对齐的，那么访问它时就很容易因为bus error而core dump。

 5，堆栈溢出

不要使用大的局部变量（因为局部变量都分配在栈上），这样容易造成堆栈溢出，破坏系统的栈和堆结构，导致出现莫名其妙的错误。

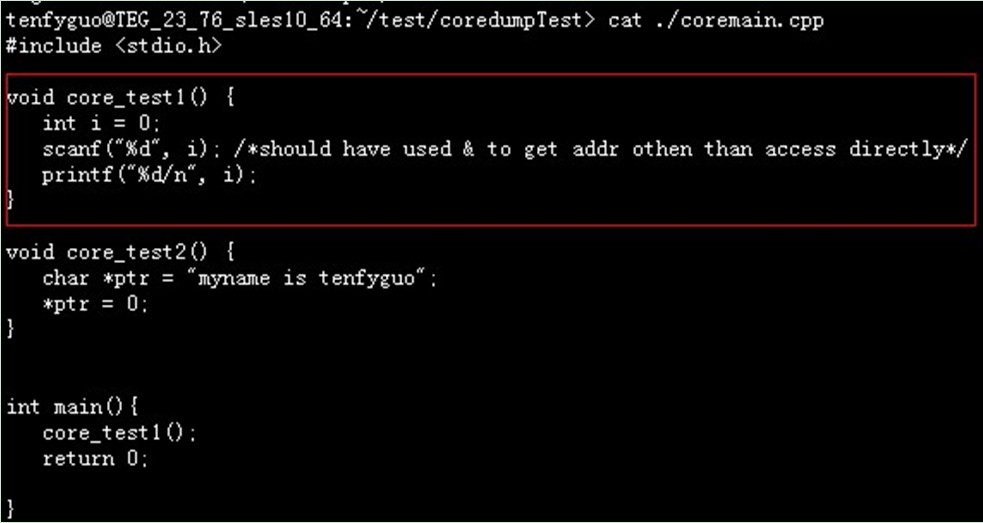
**六，利用gdb进行coredump的定位**

  其实分析coredump的工具有很多，现在大部分类unix系统都提供了分析coredump文件的工具，不过，我们经常用到的工具是gdb。

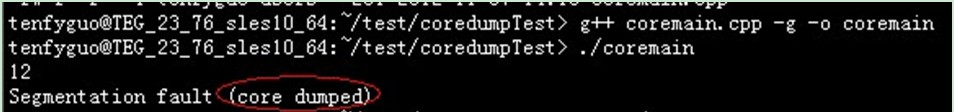
  这里我们以程序为例子来说明如何进行定位。

1，  段错误 – segmentfault

Ø  我们写一段代码往受到系统保护的地址写内容。

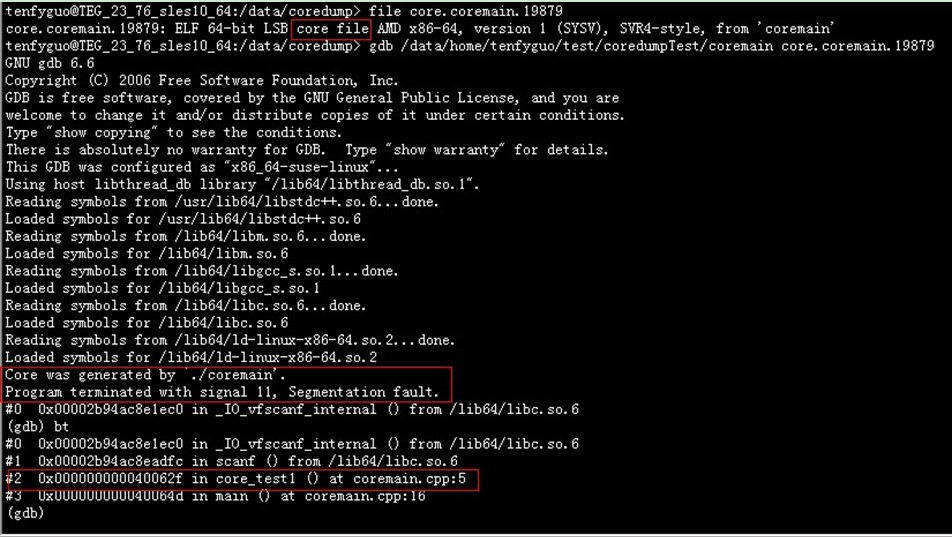


Ø  按如下方式进行编译和执行，注意这里需要-g选项编译。



可以看到，当输入12的时候，系统提示段错误并且core dumped

Ø  我们进入对应的core文件生成目录，优先确认是否core文件格式并启用gdb进行调试。



从红色方框截图可以看到，程序中止是因为信号11，且从bt(backtrace)命令（或者where）可以看到函数的调用栈，即程序执行到coremain.cpp的第5行，且里面调用scanf 函数，而该函数其实内部会调用\_IO\_vfscanf\_internal()函数。

接下来我们继续用gdb，进行调试对应的程序。

记住几个常用的gdb命令：

l(list) ，显示源代码，并且可以看到对应的行号；

b(break)x, x是行号，表示在对应的行号位置设置断点；

p(print)x, x是变量名，表示打印变量x的值

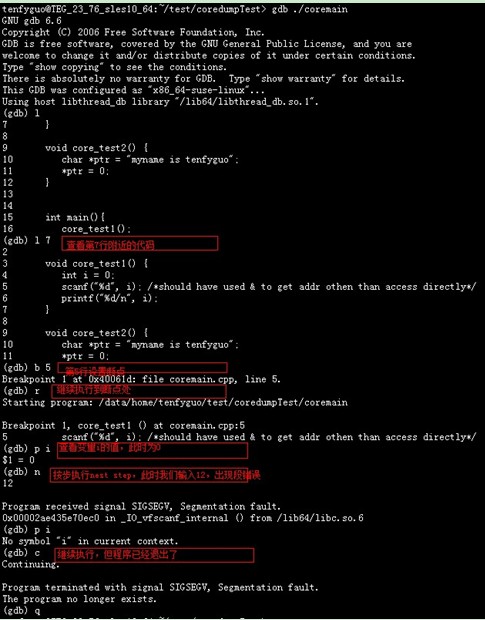
r(run), 表示继续执行到断点的位置

n(next),表示执行下一步

c(continue),表示继续执行

q(quit)，表示退出gdb

启动gdb,注意该程序编译需要-g选项进行。



注：  SIGSEGV     11       Core    Invalid memoryreference

**七，附注：**

1，  gdb的查看源码

显示源代码

GDB 可以打印出所调试程序的源代码，当然，在程序编译时一定要加上-g的参数，把源程序信息编译到执行文件中。不然就看不到源程序了。当程序停下来以后，GDB会报告程序停在了那个文件的第几行上。你可以用list命令来打印程序的源代码。还是来看一看查看源代码的GDB命令吧。

list<linenum>

显示程序第linenum行的周围的源程序。

list<function>

显示函数名为function的函数的源程序。

list

显示当前行后面的源程序。

list -

显示当前行前面的源程序。

一般是打印当前行的上5行和下5行，如果显示函数是是上2行下8行，默认是10行，当然，你也可以定制显示的范围，使用下面命令可以设置一次显示源程序的行数。

setlistsize <count>

设置一次显示源代码的行数。

showlistsize

查看当前listsize的设置。

list命令还有下面的用法：

list<first>, <last>

显示从first行到last行之间的源代码。

list ,<last>

显示从当前行到last行之间的源代码。

list +

往后显示源代码。

一般来说在list后面可以跟以下这些参数：

<linenum>   行号。

<+offset>   当前行号的正偏移量。

<-offset>   当前行号的负偏移量。

<filename:linenum>  哪个文件的哪一行。

<function>  函数名。

<filename:function>哪个文件中的哪个函数。

<\*address>  程序运行时的语句在内存中的地址。

2，  一些常用signal的含义

SIGABRT：调用abort函数时产生此信号。进程异常终止。

SIGBUS：指示一个实现定义的硬件故障。

SIGEMT：指示一个实现定义的硬件故障。EMT这一名字来自PDP-11的emulator trap 指令。

SIGFPE：此信号表示一个算术运算异常，例如除以0，浮点溢出等。

SIGILL：此信号指示进程已执行一条非法硬件指令。4.3BSD由abort函数产生此信号。SIGABRT现在被用于此。

SIGIOT：这指示一个实现定义的硬件故障。IOT这个名字来自于PDP-11对于输入／输出TRAP(input/outputTRAP)指令的缩写。系统V的早期版本，由abort函数产生此信号。SIGABRT现在被用于此。

SIGQUIT：当用户在终端上按退出键（一般采用Ctrl-/）时，产生此信号，并送至前台进

程组中的所有进程。此信号不仅终止前台进程组（如SIGINT所做的那样），同时产生一个core文件。

SIGSEGV：指示进程进行了一次无效的存储访问。名字SEGV表示“段违例（segmentationviolation）”。

SIGSYS：指示一个无效的系统调用。由于某种未知原因，进程执行了一条系统调用指令，但其指示系统调用类型的参数却是无效的。

SIGTRAP：指示一个实现定义的硬件故障。此信号名来自于PDP-11的TRAP指令。

SIGXCPUSVR4和4.3+BSD支持资源限制的概念。如果进程超过了其软C P U时间限制，则产生此信号。

SIGXFSZ：如果进程超过了其软文件长度限制，则SVR4和4.3+BSD产生此信号。

3，  Core\_pattern的格式

可以在core\_pattern模板中使用变量还很多，见下面的列表：

%% 单个%字符

%p 所dump进程的进程ID

%u 所dump进程的实际用户ID

%g 所dump进程的实际组ID

%s 导致本次core dump的信号

%t core dump的时间 (由1970年1月1日计起的秒数)

%h 主机名

%e 程序文件名