反爬虫技术和反抢购作弊在原理上是类似的，都是为了禁止机器自动操作。

以下是总结的一些反爬虫的手段，同样适用于反抢购作弊：

1.通过爬虫的特有行为模式来发现爬虫：（链接并发度，访问频率，访问数据的范围）

用netstat检查80端口的连接:

sh netstat -nt | grep youhostip:80 | awk '{print $5}' | awk -F":" '{print $1}'| sort | uniq -c | sort -r -n

这行shell可以按照80端口连接数量对来源IP进行排序，这样可以直观的判断出来网页爬虫。一般来说爬虫的并发连接非常高。 直接封锁爬虫所在的C网段地址。这是因为一般爬虫都是运行在托管机房里面，可能在一个C段里面的多台服务器上面都有爬虫，而这个C段不可能是用户宽带上网，封锁C段可以很大程度上解决问题。

频率限制，每分钟超过N次访问的IP封掉，如3小时访问量超过50次时，弹出验证框，验证内容输入错误，则计入黑名单

爬虫爬取网页的频率都是比较固定的，不像人去访问网页，中间的间隔时间比较无规则，所以我们可以给每个IP地址建立一个时间窗口，记录IP地址最近12次访问时间，每记录一次就滑动一次窗口，比较最近访问时间和当前时间，

如果间隔时间很长判断不是爬虫，清除时间窗口，如果间隔不长，就回溯计算指定时间段的访问频率，如果访问频率超过阀值，就转向验证码页面让用户填写验证码

爬虫一般不抓取css、js等资源文件

如果是垂直爬虫，往往访问数据具有明显的目的性，如果返现某些IP访问某些数据超过正常频次，且这些这些IP并没有访问其他任何资源，则可以断定是爬虫

搞个计数器，把特定ip一天内的总次数和每秒内的频率记录下来，达到实时拦截的目的（这个类似淘宝的TMD，用分布式缓存计数，超出规则，要么直接拒绝，要么跳到输入验证码的页面）

把apache或nginx日志分析和js埋点对比来分析，爬虫请求在无法模拟浏览器请求的时候，无法执行js代码，页面没有埋点，但是这个请求，会在web日志中留下，js有小问题，就是用户请求的时候，如果浏览器有缓存，实际上没有请求服务器端，但是js确埋了一个点，对比分析，效果会好很多

诱捕

比如用一些人类看不到的链接，让爬虫进去，然后封禁对应ip。

使爬虫无法识别内容

是把链接变成其他标签，用js监听点击、键盘等事件，然后跳转，爬虫不具备这样的交互能力，就找不到了。或者全站用ajax，反seo。或者把部分文字变成图片，这是百度反360爬虫的策略，把能破坏语义的高频文字（比如的、是、不等）替换成图片，不影响人类阅读。

利用http请求头部信息

useragent=JavaClient1.6（HttpClient的爬虫）

同一个IP有使用过多个useragent

请求是否含有refer:refer表示该请求从哪里跳转而来，很多爬虫没有这个字段

利用cookie，每次请求重新种一个，下次请求验证这个cookie；或者定时更改cookie为了避免误伤某些搜索引擎的爬虫，例如将百度等爬虫设置在白名单中，通过hostname检测请求IP是否属于百度

手工识别和拒绝爬虫的访问

相当多的爬虫对网站会造成非常高的负载，因此识别爬虫的来源IP是很容易的事情。最简单的办法就是用netstat检查80端口的连接：

netstat -nt | grep youhostip:80 | awk '{print $5}' | awk -F":" '{print $1}'| sort | uniq -c | sort -r -n

这行shell可以按照80端口连接数量对来源IP进行排序，这样可以直观的判断出来网页爬虫。一般来说爬虫的并发连接非常高。

如果使用lighttpd做Web Server，那么就更简单了。lighttpd的mod\_status提供了非常直观的并发连接的信息，包括每个连接的来源IP，访问的URL，连接状态和连接时间等信息，只要检查那些处于handle-request状态的高并发IP就可以很快确定爬虫的来源IP了。

拒绝爬虫请求既可以通过内核防火墙来拒绝，也可以在web server拒绝，比方说用iptables拒绝：

iptables -A INPUT -i eth0 -j DROP -p tcp --dport 80 -s 84.80.46.0/24

直接封锁爬虫所在的C网段地址。这是因为一般爬虫都是运行在托管机房里面，可能在一个C段里面的多台服务器上面都有爬虫，而这个C段不可能是用户宽带上网，封锁C段可以很大程度上解决问题。

通过识别爬虫的User-Agent信息来拒绝爬虫

有很多爬虫并不会以很高的并发连接爬取，一般不容易暴露自己；有些爬虫的来源IP分布很广，很难简单的通过封锁IP段地址来解决问题；另外还有很多各种各样的小爬虫，它们在尝试Google以外创新的搜索方式，每个爬虫每天爬取几万的网页，几十个爬虫加起来每天就能消耗掉上百万动态请求的资源，由于每个小爬虫单独的爬取量都很低，所以你很难把它从每天海量的访问IP地址当中把它准确的挖出来。

这种情况下我们可以通过爬虫的User-Agent信息来识别。每个爬虫在爬取网页的时候，会声明自己的User-Agent信息，因此我们就可以通过记录和分析User-Agent信息来挖掘和封锁爬虫。我们需要记录每个请求的User-Agent信息，对于Rails来说我们可以简单的在app/controllers/application.rb里面添加一个全局的before\_filter，来记录每个请求的User-Agent信息：

logger.info "HTTP\_USER\_AGENT #{request.env["HTTP\_USER\_AGENT"]}"

然后统计每天的production.log，抽取User-Agent信息，找出访问量最大的那些User-Agent。要注意的是我们只关注那些爬虫的User-Agent信息，而不是真正浏览器User-Agent，所以还要排除掉浏览器User-Agent，要做到这一点仅仅需要一行shell：

grep HTTP\_USER\_AGENT production.log | grep -v -E 'MSIE|Firefox|Chrome|Opera|Safari|Gecko' | sort | uniq -c | sort -r -n | head -n 100 > bot.log

统计结果类似这样：

57335 HTTP\_USER\_AGENT Baiduspider+(+http://www.baidu.com/search/spider.htm)

56639 HTTP\_USER\_AGENT Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)

42610 HTTP\_USER\_AGENT Mediapartners-Google

19131 HTTP\_USER\_AGENT msnbot/2.0b (+http://search.msn.com/msnbot.htm)

从日志就可以直观的看出每个爬虫的请求次数。要根据User-Agent信息来封锁爬虫是件很容易的事情，lighttpd配置如下：

$HTTP["useragent"] =~ "qihoobot|^Java|Commons-HttpClient|Wget|^PHP|Ruby|Python" {

url.rewrite = ( "^/(.\*)" => "/crawler.html" )

}

使用这种方式来封锁爬虫虽然简单但是非常有效，除了封锁特定的爬虫，还可以封锁常用的编程语言和HTTP类库的User-Agent信息，这样就可以避免很多无谓的程序员用来练手的爬虫程序对网站的骚扰。

还有一种比较常见的情况，就是某个搜索引擎的爬虫对网站爬取频率过高，但是搜索引擎给网站带来了很多流量，我们并不希望简单的封锁爬虫，仅仅是希望降低爬虫的请求频率，减轻爬虫对网站造成的负载，那么我们可以这样做：

$HTTP["user-agent"] =~ "Baiduspider+" {

connection.delay-seconds = 10

}

对百度的爬虫请求延迟10秒钟再进行处理，这样就可以有效降低爬虫对网站的负载了。

通过网站流量统计系统和日志分析来识别爬虫

有些爬虫喜欢修改User-Agent信息来伪装自己，把自己伪装成一个真实浏览器的User-Agent信息，让你无法有效的识别。这种情况下我们可以通过网站流量系统记录的真实用户访问IP来进行识别。

主流的网站流量统计系统不外乎两种实现策略：一种策略是在网页里面嵌入一段js，这段js会向特定的统计服务器发送请求的方式记录访问量；另一种策略是直接分析服务器日志，来统计网站访问量。在理想的情况下，嵌入js的方式统计的网站流量应该高于分析服务器日志，这是因为用户浏览器会有缓存，不一定每次真实用户访问都会触发服务器的处理。但实际情况是，分析服务器日志得到的网站访问量远远高于嵌入js方式，极端情况下，甚至要高出10倍以上。

现在很多网站喜欢采用awstats来分析服务器日志，来计算网站的访问量，但是当他们一旦采用Google Analytics来统计网站流量的时候，却发现GA统计的流量远远低于awstats，为什么GA和awstats统计会有这么大差异呢？罪魁祸首就是把自己伪装成浏览器的网络爬虫。这种情况下awstats无法有效的识别了，所以awstats的统计数据会虚高。

其实作为一个网站来说，如果希望了解自己的网站真实访问量，希望精确了解网站每个频道的访问量和访问用户，应该用页面里面嵌入js的方式来开发自己的网站流量统计系统。自己做一个网站流量统计系统是件很简单的事情，写段服务器程序响应客户段js的请求，分析和识别请求然后写日志的同时做后台的异步统计就搞定了。

通过流量统计系统得到的用户IP基本是真实的用户访问，因为一般情况下爬虫是无法执行网页里面的js代码片段的。所以我们可以拿流量统计系统记录的IP和服务器程序日志记录的IP地址进行比较，如果服务器日志里面某个IP发起了大量的请求，在流量统计系统里面却根本找不到，或者即使找得到，可访问量却只有寥寥几个，那么无疑就是一个网络爬虫。

分析服务器日志统计访问最多的IP地址段一行shell就可以了：

grep Processing production.log | awk '{print $4}' | awk -F'.' '{print $1"."$2"."$3".0"}' | sort | uniq -c | sort -r -n | head -n 200 > stat\_ip.log

然后把统计结果和流量统计系统记录的IP地址进行对比，排除真实用户访问IP，再排除我们希望放行的网页爬虫，比方Google，百度，微软msn爬虫等等。最后的分析结果就就得到了爬虫的IP地址了。以下代码段是个简单的实现示意：

whitelist = []

IO.foreach("#{RAILS\_ROOT}/lib/whitelist.txt") { |line| whitelist << line.split[0].strip if line }

realiplist = []

IO.foreach("#{RAILS\_ROOT}/log/visit\_ip.log") { |line| realiplist << line.strip if line }

iplist = []

IO.foreach("#{RAILS\_ROOT}/log/stat\_ip.log") do |line|

ip = line.split[1].strip

iplist << ip if line.split[0].to\_i > 3000 && !whitelist.include?(ip) && !realiplist.include?(ip)

end

Report.deliver\_crawler(iplist)

分析服务器日志里面请求次数超过3000次的IP地址段，排除白名单地址和真实访问IP地址，最后得到的就是爬虫IP了，然后可以发送邮件通知管理员进行相应的处理。

网站的实时反爬虫防火墙实现策略

通过分析日志的方式来识别网页爬虫不是一个实时的反爬虫策略。如果一个爬虫非要针对你的网站进行处心积虑的爬取，那么他可能会采用分布式爬取策略，比方说寻找几百上千个国外的代理服务器疯狂的爬取你的网站，从而导致网站无法访问，那么你再分析日志是不可能及时解决问题的。所以必须采取实时反爬虫策略，要能够动态的实时识别和封锁爬虫的访问。

要自己编写一个这样的实时反爬虫系统其实也很简单。比方说我们可以用memcached来做访问计数器，记录每个IP的访问频度，在单位时间之内，如果访问频率超过一个阀值，我们就认为这个IP很可能有问题，那么我们就可以返回一个验证码页面，要求用户填写验证码。如果是爬虫的话，当然不可能填写验证码，所以就被拒掉了，这样很简单就解决了爬虫问题。

用memcache记录每个IP访问计数，单位时间内超过阀值就让用户填写验证码，用Rails编写的示例代码如下：

ip\_counter = Rails.cache.increment(request.remote\_ip)

if !ip\_counter

Rails.cache.write(request.remote\_ip, 1, :expires\_in => 30.minutes)

elsif ip\_counter > 2000

render :template => 'test', :status => 401 and return false

end

这段程序只是最简单的示例，实际的代码实现我们还会添加很多判断，比方说我们可能要排除白名单IP地址段，要允许特定的User-Agent通过，要针对登录用户和非登录用户，针对有无referer地址采取不同的阀值和计数加速器等等。

此外如果分布式爬虫爬取频率过高的话，过期就允许爬虫再次访问还是会对服务器造成很大的压力，因此我们可以添加一条策略：针对要求用户填写验证码的IP地址，如果该IP地址短时间内继续不停的请求，则判断为爬虫，加入黑名单，后续请求全部拒绝掉。为此，示例代码可以改进一下：

before\_filter :ip\_firewall, :except => :test

def ip\_firewall

render :file => "#{RAILS\_ROOT}/public/403.html", :status => 403 if BlackList.include?(ip\_sec)

end

我们可以定义一个全局的过滤器，对所有请求进行过滤，出现在黑名单的IP地址一律拒绝。对非黑名单的IP地址再进行计数和统计：

ip\_counter = Rails.cache.increment(request.remote\_ip)

if !ip\_counter

Rails.cache.write(request.remote\_ip, 1, :expires\_in => 30.minutes)

elsif ip\_counter > 2000

crawler\_counter = Rails.cache.increment("crawler/#{request.remote\_ip}")

if !crawler\_counter

Rails.cache.write("crawler/#{request.remote\_ip}", 1, :expires\_in => 10.minutes)

elsif crawler\_counter > 50

BlackList.add(ip\_sec)

render :file => "#{RAILS\_ROOT}/public/403.html", :status => 403 and return false

end

render :template => 'test', :status => 401 and return false

end

如果某个IP地址单位时间内访问频率超过阀值，再增加一个计数器，跟踪他会不会立刻填写验证码，如果他不填写验证码，在短时间内还是高频率访问，就把这个IP地址段加入黑名单，除非用户填写验证码激活，否则所有请求全部拒绝。这样我们就可以通过在程序里面维护黑名单的方式来动态的跟踪爬虫的情况，甚至我们可以自己写个后台来手工管理黑名单列表，了解网站爬虫的情况。

这个策略已经比较智能了，但还可以继续改进：

1、用网站流量统计系统来改进实时反爬虫系统

网站流量统计系统记录的IP地址是真实用户访问IP，所以我们在网站流量统计系统里面也去操作memcached，但是这次不是增加计数值，而是减少计数值。在网站流量统计系统里面每接收到一个IP请求，就相应的cache.decrement(key)。所以对于真实用户的IP来说，它的计数值总是加1然后就减1，不可能很高。这样我们就可以大大降低判断爬虫的阀值，可以更加快速准确的识别和拒绝掉爬虫。

2、用时间窗口来改进实时反爬虫系统

爬虫爬取网页的频率都是比较固定的，不像人去访问网页，中间的间隔时间比较无规则，所以我们可以给每个IP地址建立一个时间窗口，记录IP地址最近12次访问时间，每记录一次就滑动一次窗口，比较最近访问时间和当前时间，如果间隔时间很长判断不是爬虫，清除时间窗口，如果间隔不长，就回溯计算指定时间段的访问频率，如果访问频率超过阀值，就转向验证码页面让用户填写验证码。

最终这个实时反爬虫系统就相当完善了，它可以很快的识别并且自动封锁爬虫的访问，保护网站的正常访问。不过有些爬虫可能相当狡猾，它也许会通过大量的爬虫测试来试探出来你的访问阀值，以低于阀值的爬取速度抓取你的网页，因此我们还需要辅助第3种办法，用日志来做后期的分析和识别，就算爬虫爬的再慢，它累计一天的爬取量也会超过你的阀值被你日志分析程序识别出来。

总之我们综合运用上面的四种反爬虫策略，可以很大程度上缓解爬虫对网站造成的负面影响，保证网站的正常访问。