# consul 文档

## 安装

## 更新

## 词汇表

此页面收集了Consul和Consul Enterprise文档中使用的一些技术术语的简要定义，以及Consul社区中交流中经常出现的一些术语。

### Agent

代理是Consul集群的每个成员上的长时间运行守护程序。 它是通过运行consul agent来启动的。 代理能够以客户端或服务器模式运行。 由于所有节点都必须运行代理，因此将节点称为客户端或服务器更简单，但代理还有其他实例。 所有代理都可以运行DNS或HTTP接口，并负责运行检查并保持服务同步。

### Client

客户端是将所有RPC转发到服务器的代理。客户端相对来说是无状态的。客户端执行的唯一后台活动是参与局域网（LAN）gossip池。这具有最小的资源开销，并且只消耗少量的网络带宽。

### Server

服务器是一种具有扩展职责集的代理，包括参与Raft 仲裁、维护集群状态、响应RPC查询、与其他数据中心（datacenters）交换WAN流言（gossip）以及将查询转发给leader或远程数据中心。

### Datacenter

我们将数据中心定义为一个私有、低延迟和高带宽的网络环境。这不包括穿越公共互联网的通信，但就我们的目的而言，一个EC2区域内的多个可用区域将被视为单个数据中心的一部分。

**注释：**亚马逊EC2

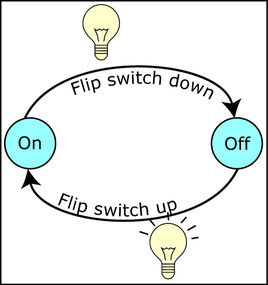
*亚马逊弹性计算云（EC2，Elastic Compute Cloud）是一个让使用者可以租用云端电脑运行所需应用的系统。EC2借由提供Web服务的方式让使用者可以弹性地运行自己的Amazon机器映像档，使用者将可以在这个虚拟机器上运行任何自己想要的软件或应用程式。提供可调整的云计算能力。它旨在使开发者的网络规模计算变得更为容易。*

*--来自百度百科*

### Consensus（协商一致/共识）

在我们的文档中使用时，我们使用共识来表示**就选举的领导者（elected leader）以及对交易的顺序**达成一致。 由于这些事务应用于有限状态机（[finite-state machine](https://en.wikipedia.org/wiki/Finite-state_machine)），因此我们对共识的定义意味着复制状态机的一致性。 维基百科（[Wikipedia](https://en.wikipedia.org/wiki/Consensus_(computer_science))）上对共识进行了更加详细的描述，我们的实现描述在[此处](https://www.consul.io/docs/internals/consensus.html)。

**注释：**有限状态机

*有限状态机，（英语：Finite-state machine, FSM），又称有限状态自动机，简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。*

*--来自百度百科*

### Gossip

Consul建立在[Serf](https://www.serf.io/)之上，它提供了一种用于多种用途的完整的gossip协议（[gossip protocol](https://en.wikipedia.org/wiki/Gossip_protocol)）。 Serf提供会员资格，故障检测和事件广播的功能。 在gossip文档（ [gossip documentation](https://www.consul.io/docs/internals/gossip.html)）种我们有更加详尽的描述。 在这里，只要知道gossip协议用于节点到节点（node-to-node）的通讯就已经足够，其主要是使用UDP协议。

### LAN Gossip

指局域网（LAN）gossip池，其中包含全部位于同一局域网或数据中心的节点。

**注意：**既包含server也包含client

### WAN Gossip

指仅包含服务器（server）的WAN gossip池。 这些服务器主要位于不同的数据中心，通常通过互联网或广域网进行通信。

### RPC

远程过程调用（**R**emote **P**rocedure **C**all）。 这是一种允许客户端向服务器发送请求的请求/响应机制。

## Consul Internals（内部结构）

本节介绍了consul 的一些内部细节。 理解Consul的内部结构对于在生产中成功使用它是必要的。

请查看以下文档以了解Consul的工作原理。

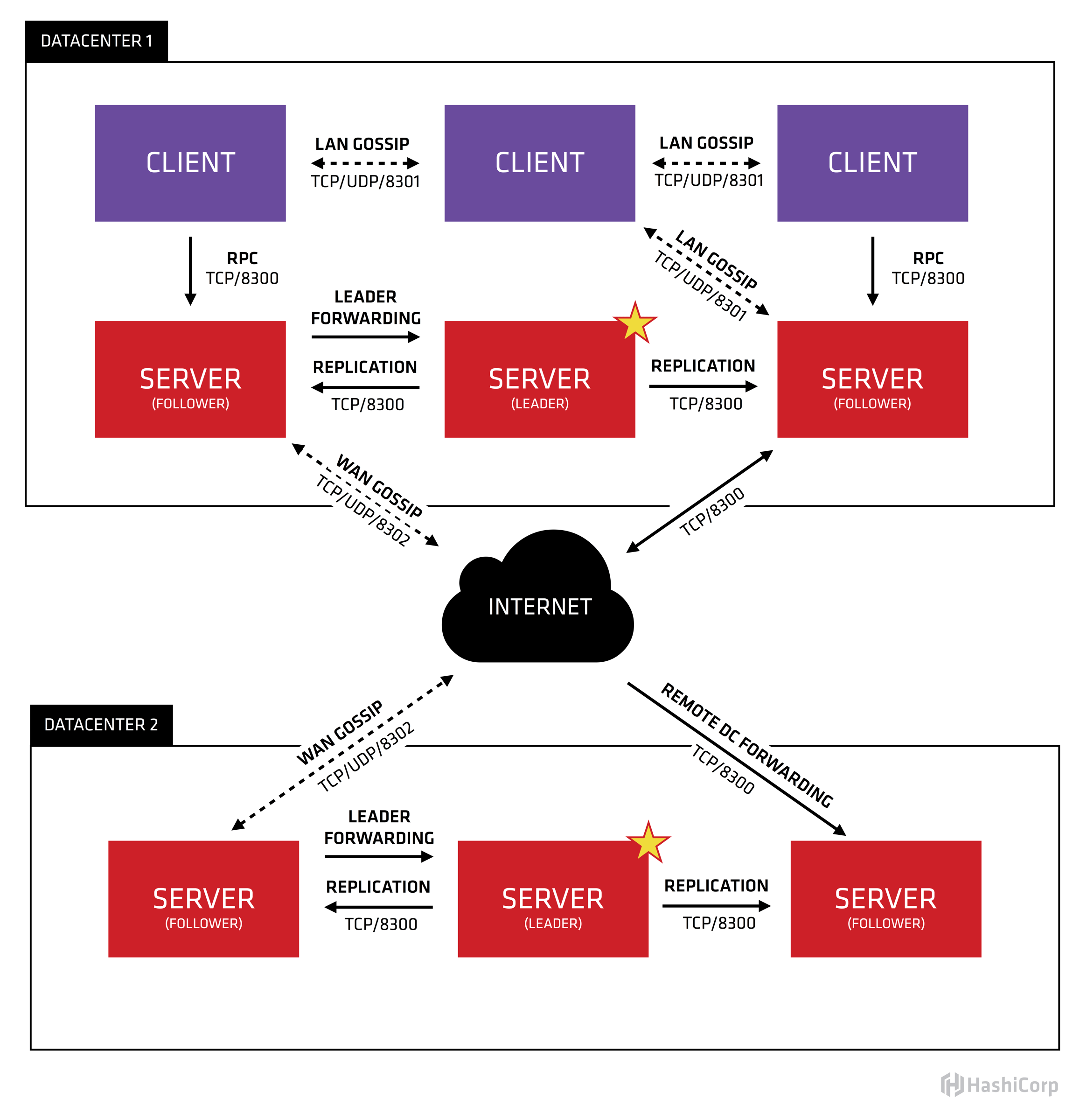
### [Architecture](https://www.consul.io/docs/internals/architecture.html)

Consul是一个复杂的系统，有许多不同的移动（moving）部件。 为了帮助Consul的用户和开发人员形成其工作原理的心理模型，该文档描述其系统架构。

**在讨论这个架构之前，我们建议阅读词汇表部分，这样对所讨论的内容将更有帮助。**

#### 10,000 foot view（总体预览）

从10,000英尺的高度，Consul的架构看起来像这样：

[](https://www.consul.io/assets/images/consul-arch-420ce04a.png)

让我们分解这个图像并描述每一块。 首先，我们可以看到有两个数据中心，标记为“数据中心1”和“数据中心2”。 Consul为多个数据中心提供一流的支持，并期望这是常见的情况。

在每个数据中心，我们拥有一个客户端和服务器和混合体。预计将有3至5个服务器（server）。这是在失败和可用性之间取得平衡，因为随着增加更多的机器，协商一致逐渐变慢。然而，客户端的数量是没有限制的，他们可以很容易地扩展到成千上万的客户端（client）。

数据中心中的所有代理（agent）都参与了gossip协议。这意味着有一个gossip池，它包含一个给定数据中心的所有代理（agent）。这有几个目的：首先，不需要使用服务器地址配置客户端；发现是自动完成的。第二，检测代理故障的工作不是放在server上，而是分布式的。这使得故障检测比幼稚的心跳方案更具可伸缩性。它还为节点提供故障检测；如果agent不可达，则该节点可能已经发生故障。第三，它被用作一个消息层，用于在发生领导人选举（leader election）等重要事件时发出通知。

**注意：**连接类型与连接个数

每个数据中心中的server都是单个Raft对等集的一部分。 这意味着他们共同选举一个领导者（leader），这个选择的server具有额外职责。领导者（leader）负责处理所有查询和交易。 作为共识协议（[consensus protocol](https://www.consul.io/docs/internals/consensus.html)）的一部分，还必须将事务复制到所有对等体。 由于此要求，当非领导者（non-leader）服务器收到RPC请求时，它会将其转发给群集leader。

服务器代理（server agent）也作为WAN gossip池的一部分运行。 此池与LAN池不同，因为它针对较高的Internet延迟进行了优化，并且预计仅包含其他Consul服务器代理（server agent）。 此池的目的是允许数据中心以低触摸方式发现彼此。 在线创建新的数据中心就像加入现有的WAN gossip 池一样简单。 由于服务器（server）都在此池中运行，因此它还支持跨数据中心请求。 当服务器（server）收到对不同数据中心的请求时，它会将其转发到正确数据中心的随机服务器（server）。 然后该服务器（server）可以转发给本地领导者（leader）。

这导致数据中心之间的耦合非常低，但由于故障检测，连接缓存和多路复用，跨数据中心请求相对快速且可靠。

通常，不会在不同的Consul数据中心之间复制数据。 当对另一个数据中心中的资源发出请求时，本地Consul服务器（server）会将RPC请求转发给该资源的远程Consul服务器（server）并返回结果。 **如果远程数据中心不可用，那么这些资源也将不可用**，但这不会影响本地数据中心。 在某些特殊情况下，可以复制有限的数据子集，例如使用Consul的内置ACL复制（[ACL replication](https://learn.hashicorp.com/consul/day-2-operations/acl-replication)）功能，或者像[consul-replicate](https://github.com/hashicorp/consul-replicate)这样的外部工具。

在某些地方，客户端代理可能会缓存来自服务器的数据，以便在本地提供数据以提高性能和可靠性。示例包括连接证书和意图（intentions），它们允许客户端代理在不需要往返服务器的情况下对入站连接请求进行本地决策。一些API端点还支持可选的结果缓存。这有助于可靠性，因为本地代理可以继续响应某些查询，如服务发现或从缓存连接授权，即使与服务器的连接中断或服务器暂时不可用。

#### Getting in depth（深入）

在这一点上，我们已经介绍了Consul的高级架构，但每个子系统还有更多细节。 共识协议（[consensus protocol](https://www.consul.io/docs/internals/consensus.html)）与gossip协议（[gossip protocol](https://www.consul.io/docs/internals/gossip.html)）同样有详细的文档说明。 此外还有其使用的安全模型和协议的文档说明。

有关其他详细信息，请查阅源代码，在IRC中询问或联系邮件列表。

### [Consensus Protocol](https://www.consul.io/docs/internals/consensus.html)

Consul使用共识协议来提供一致性（由CAP定义）。 共识协议基于“Raft：寻找可理解的共识算法”。 有关Raft的直观解释，请参阅数[The Secret Lives of Data](http://thesecretlivesofdata.com/raft)。

#### Raft Protocol Overview（Raft 协议）

Raft是一种基于[Paxos](https://en.wikipedia.org/wiki/Paxos_%28computer_science%29)的共识算法。 与Paxos相比，Raft被设计为具有更少的状态和更简单，更易理解的算法。

讨论Raft时需要了解一些关键术语：

* 日志 - Raft系统中的主要工作单元是日志条目。 一致性问题可以分解到复制日志中。 日志是有序的条目序列。 条目包括任何群集更改：添加节点，添加服务，新的键值对等。如果所有成员都同意条目及其顺序，我们认为日志是一致的。
* 有限状态机 - [Finite State Machine](https://en.wikipedia.org/wiki/Finite-state_machine) FSM是有限状态的集合，它们之间可以互相转换。 在应用新日志时，允许FSM在状态之间转换。 应用相同的日志序列必须导致相同的状态，这意味着行为必须是确定性的。
* 对等集 -Peer set -对等集是参与日志复制的所有成员的集合。 出于Consul的目的，所有服务器节点都位于本地数据中心的对等集中。
* 仲裁Quorum  -仲裁是来自对等集的大多数成员：对于一组大小为n，仲裁需要至少（n / 2）+ 1个成员。 例如，如果对等集中有5个成员，则需要3个节点才能形成仲裁。 如果仲裁数量的节点因任何原因不可用，则群集将变为不可用，并且不能提交新日志。
* 持久条目Committed Entry -当条目持久地存储在仲裁要求数量的节点上时，将认为该条目已提交。 一旦提交了条目，就可以应用它。
* Leader -在任何给定时间，对等集选择单个节点作为领导者。 领导者负责摄取新的日志条目，复制到关注者，以及管理何时认为条目被提交。

### [Gossip Protocol](https://www.consul.io/docs/internals/gossip.html)

### [Network Coordinates](https://www.consul.io/docs/internals/coordinates.html)

### [Sessions](https://www.consul.io/docs/internals/sessions.html)

### [Anti-Entropy](https://www.consul.io/docs/internals/anti-entropy.html)

Consul使用一种先进的服务和健康信息维护方法。本页详细介绍如何注册服务和检查，如何填充目录，以及如何在服务和检查更改时更新健康状态信息。

#### Components

首先了解服务和健康检查中涉及的移动部分是很重要的：代理（agent）和目录（catalog）。为了使反熵更容易理解，下面对这些进行了概念描述。

##### Agent

每个Consul agent 维护着他所拥有的**服务集合**（service）和**检查注册**同时还有**健康信息**。agent负责执行他所拥有的健康检查并更新他们的本地状态。

在agent上下文中的Services和checks拥有丰富的配置项集合可用。这是因为agent负责通过使用健康检查（health checks）来生成他的服务和以及他们的健康信息。

##### Catalog

Consul的服务发现由服务目录支持。 该目录是通过汇总**Agent提交的信息**而形成的。 目录维护群集的高级视图，包括**可用的服务**，运行这些**服务的节点**，**运行状况**信息等。 该目录用于通过Consul提供的各种接口（包括DNS和HTTP）公开此信息。

与Agent相比，Catalog上下文中的服务和检查具有更有限的字段集。 这是因为目录仅负责记录和返回有关服务，节点和运行状况的信息。

目录仅由服务器节点维护。 这是因为通过Raft日志复制目录，以提供集群的统一且一致的视图。

#### Anti-Entropy

熵是系统变得越来越无序的趋势。 Consul的反熵机制旨在抵消这种趋势，即使组件故障也能保持集群的状态。

如上所述，Consul在全局服务目录和代理的本地状态之间有明确的区分。 反熵机制协调了这两个世界观：反熵是本地代理状态和目录的同步。 例如，当用户注册新服务或与代理核对时，代理会依次通知目录该新检查存在。 同样，当从代理中删除检查时，也会从目录中删除它。

反熵还用于更新可用性信息。 **当代理运行其运行状况检查时**(Agent 本身的健康)，它们的状态可能会发生变化，在这种情况下，它们的新状态将同步到目录。 使用此信息，目录可以根据其可用性智能地响应有关其节点和服务的查询。

在此同步过程中，还将检查目录是否正确。如果目录中存在代理不知道的任何服务或检查，则将自动删除这些服务或检查，以使目录反映该代理的适当服务和健康信息集。Consul将代理状态视为权威；如果代理视图与目录视图有任何差异，则将**始终使用代理-本地视图**。

#### Periodic Synchronization（ 周期同步）

除了在代理发生更改时运行，反熵也是一个长期运行的进程，它会定期唤醒以同步服务并检查目录的状态。 这可确保目录与代理的真实状态紧密匹配。 即使在完全数据丢失的情况下，这也允许Consul重新填充服务目录。

为了避免饱和，周期性反熵运行之间的时间量将根据集群大小而变化。 下表定义了集群大小和同步间隔之间的关系：

|  |  |
| --- | --- |
| **Cluster Size** | **Periodic Sync Interval** |
| 1 - 128 | 1 minute |
| 129 - 256 | 2 minutes |
| 257 - 512 | 3 minutes |
| 513 - 1024 | 4 minutes |
| ... | ... |

以上间隔是近似值。 每个Consul代理将在间隔窗口内选择随机交错的开始时间，以避免雷鸣般的群体。

#### Best-effort sync（尽力同步）

在许多情况下，反熵可能会失败，包括代理配置错误或其操作环境，I / O问题（完整磁盘，文件系统权限等），网络问题（代理无法与服务器通信）等。 因此，代理尝试以尽力而为的方式进行同步。

如果在反熵运行期间遇到错误，则会记录错误并且代理继续运行。 定期运行反熵机制以自动从这些类型的瞬态故障中恢复。

#### Enable Tag Override

可以部分修改服务注册的同步，以允许外部代理更改服务的标记。 这在外部监控服务需要成为标签信息真实来源的情况下非常有用。 例如，Redis数据库及其监控服务Redis Sentinel具有这种关系。 Redis实例负责其大部分配置，但Sentinels确定Redis实例是主实例还是辅助实例。 使用Consul服务配置项enable\_tag\_override，您可以指示运行Redis数据库的Consul代理在反熵同步期间不更新标记。 有关更多信息，请参阅服务页面

### [Security Model](https://www.consul.io/docs/internals/security.html)

在部署生产数据中心之前，您还应该熟悉Jepsen测试（[Jepsen testing](https://www.consul.io/docs/internals/jepsen.html)）。

**注释：**Jepsen测试（吉布森测试）

[jepsen](https://link.jianshu.com/?t=https://github.com/aphyr/jepsen)是一个分布式测试库，我们可以使用它对某个分布式系统执行一系列操作，并最终验证这些操作是否正确执行

## Consul Commands (CLI)

# HTTP API Structure HTTP API 结构

Consul的主要接口是RESTful HTTP API。 API可以对节点，服务，检查，配置等执行基本的CRUD操作。

## Authentication 认证

启用身份验证后，应使用X-Consul-Token标头或授权标头中的承载方案向API请求提供Consul令牌。 这降低了令牌意外被记录或暴露的可能性。 使用身份验证时，客户端应通过TLS进行通信。 如果您未在请求中提供令牌，则将**使用代理默认令牌**。

下面是使用带有X-Consul-Token的curl的示例。

$ curl \

--header "X-Consul-Token: <consul token>" \

http://127.0.0.1:8500/v1/agent/members

下面是使用带有承载方案的curl的示例。

$ curl \

--header "Authorization: Bearer <consul token>" \

http://127.0.0.1:8500/v1/agent/members

以前这是通过？token = query参数提供的。 此功能存在于许多端点上以实现向后兼容，但强烈建议不要使用此功能，因为它可以作为URL的一部分显示在访问日志中。

要了解有关ACL系统的更多信息，请阅读文档。

## Version Prefix 版本前缀

所有API路由都以/ v1 /为前缀。 本文档仅适用于v1 API。

## Formatted JSON Output --格式化JSON输出

默认情况下，所有HTTP API请求的输出都是最小化的JSON。 如果客户端在查询字符串上传递相同内容，则将返回格式化的JSON。

## HTTP Methods HTTP 方法

Consul的API旨在成为RESTful，尽管有一些例外。 API响应标准HTTP谓词GET，PUT和DELETE。 每个API方法都会清楚地记录它响应的动词和生成的响应。 具有不同动词的相同路径可以触发不同的行为。 例如：

PUT /v1/kv/foo

GET /v1/kv/foo

即使这些共享路径，PUT操作也会创建新密钥，而GET操作会读取现有密钥。

以下是使用curl的相同示例：

$ curl \

--request PUT \

--data 'hello consul' \

http://127.0.0.1:8500/v1/kv/foo

## Translated Addresses 转换地址

Consul 0.7添加了根据translate\_wan\_addrs的配置设置转换HTTP响应中的地址的功能。 为了使客户端能够知道地址转换是否有效，如果启用了转换，将添加X-Consul-Translate-Addresses标头，其值为true。 如果未启用转换，则此标头将不存在。

## UUID Format

Consul API生成的UUID格式标识符使用hashicorp / go-uuid库。

这些UUID格式字符串是使用高质量，纯随机字节生成的。 它不打算符合RFC，只是使用128位值的易于理解的字符串表示。

# KV Store Endpoints(KV存储终端)

/kv端点访问Consul的简单键/值存储，对存储服务配置或其他元数据很有用。

请务必注意，每个数据中心都有自己的KV存储，并且数据中心之间没有内置复制。 如果您对数据中心之间的复制感兴趣，请查看Consul Replicate项目。

KV存储库中的值不能大于512Kb

对于多键更新，请考虑使用**事务**。

## Read Key

终端将返回指定的key，如果key再指定的路径上不存在，则返回404而不是200的响应。

对于多key读取，请考虑使用事务

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| GET | /kv/:key | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/features/blocking.html), [consistency modes](https://www.consul.io/api/features/consistency.html), [agent caching](https://www.consul.io/api/features/caching.html), and [required ACLs](https://www.consul.io/api/index.html#authentication).

如下表显示了终端对阻塞查询，一致性模型，agent缓存，ACL依赖的支持

| **Blocking Queries** | **Consistency Modes** | **Agent Caching** | **ACL Required** |
| --- | --- | --- | --- |
| YES | all | none | key:read |

### 参数

* [key](https://www.consul.io/api/kv.html#key) (string: "") -指定读取key的路径。

* [dc](https://www.consul.io/api/kv.html" \l "dc) (string: "") -指定查询的数据中心。默认是agent正在查询的数据中心。这可以作为一个查询参数如URL的一部分。

* [recurse](https://www.consul.io/api/kv.html" \l "recurse) (bool: false) -指定查找是否应该是递归的，并将键视为前缀而不是文字匹配。这可以作为一个查询参数如URL的一部分。
* [raw](https://www.consul.io/api/kv.html#raw) (bool: false) -指定响应只是键的原始值，没有任何编码或元数据。这可以作为一个查询参数如URL的一部分。
* [keys](https://www.consul.io/api/kv.html#keys) (bool: false) -指定仅返回键（无值或元数据）。这可以作为一个查询参数如URL的一部分。
* [separator](https://www.consul.io/api/kv.html#separator) (string: '') -指定用作递归键查找的分隔符的字符串。 此选项仅在与keys参数配对时使用，以限制返回的键的前缀，仅限于给定的分隔符。这可以作为一个查询参数如URL的一部分。

### Sample Request

$ curl \

http://127.0.0.1:8500/v1/kv/my-key

### Sample Response

**Metadata Response**

[

{

"CreateIndex": 100,

"ModifyIndex": 200,

"LockIndex": 200,

"Key": "zip",

"Flags": 0,

"Value": "dGVzdA==",

"Session": "adf4238a-882b-9ddc-4a9d-5b6758e4159e"

}

]

* [CreateIndex](https://www.consul.io/api/kv.html#createindex) 是表示创建条目的内部索引值

* [ModifyIndex](https://www.consul.io/api/kv.html" \l "modifyindex) 是修改这个可以的最后的索引。这个索引对应于返回的响应头的X-Consul-Index 的值，并且他可以通过设置?index请求参数用于建立一个阻塞查询。你甚至可以对整个KV存储子树执行阻塞查询。如果?recurse被提供。则返回的X-Consul-Index对应于前缀内的最后ModifyIndex，并且使用 ?index的阻塞查询阻塞等待那个前缀内的key被更新才返回。

* [LockIndex](https://www.consul.io/api/kv.html" \l "lockindex) 在一个锁中成功获取key的次数。如果持有锁，这个会话key将提供拥有此锁的会话。

* [Key](https://www.consul.io/api/kv.html" \l "key-1) 只是条目简单路径。

* [Flags](https://www.consul.io/api/kv.html" \l "flags)是一个不透明的无符号整数，可以附加到每个条目。 客户可以选择使用它，但对他们的应用程序应有意义。

* [Value](https://www.consul.io/api/kv.html" \l "value) base64位编码的数据块。

**Keys Response**

使用？keys查询参数时，响应结构将更改为字符串数组而不是JSON对象数组。 列表/ web /带有/ separator可能会返回：

[

"/web/bar",

"/web/foo",

"/web/subdir/"

]

# API功能

## Consistency Modes 一致性模型

大多数读取查询端点支持多个级别的一致性。 由于没有任何策略可以满足所有客户的需求，因此这些一致性模式允许用户在如何平衡分布式系统中固有的权衡方面拥有最终的发言权。

三种读取模式是：

* default - 如果未指定，则几乎在所有情况下默认值都强一致性。 但是，有一个小窗口可以选出新的领导者（leader），在此期间老leader仍可以服务过时的值。这种权衡方式读取速度快，但可能是过时的值。 导致过时读取的条件很难触发，大多数客户端不应该担心这种情况。 另请注意，此竞争条件仅适用于读取，而不适用于写入。

要切换这些模式，应该在请求中提供陈旧的或一致的查询参数。提供两者都是错误的。

请注意，一些端点支持缓存参数，该参数的语义与陈旧的参数相同，但取舍不同。代理缓存功能文档中描述了此行为。

为了支持数据的可接受丢失，响应提供了X-Consuer-LastContact报头，其中包含了服务器最后一次被leader节点联系的时间，以毫秒为单位。X-Consul-KnownLeader 标题还表示是否存在已知的leader.客户可以使用这些信息来判断结果的损耗程度，并采取适当的行动。

## Blocking Queries

Consul中的许多端点都支持称为“阻塞查询”的功能。阻塞查询用于使用长轮询等待潜在的更改。并非所有端点都支持阻止，但每个端点都唯一地记录了它对文档中阻止查询的支持。

支持阻塞查询的端点返回名为X-Consul-Index的HTTP头。这是表示所请求资源的当前状态的唯一标识符。

在对此资源的后续请求中，客户端可以将索引查询字符串参数设置为X-Consul-Index的值，指示客户端希望等待该索引之后的任何更改。

提供此选项后，HTTP请求将“挂起”，直到系统发生更改或达到最大超时。一个关键的注意事项是，阻止请求的返回并不能保证更改。可能已达到超时或存在不影响查询结果的幂等写入。

除了索引之外，支持阻塞的端点还将遵循指定阻塞请求的最大持续时间的等待参数。这仅限10分钟。如果未设置，则等待时间默认为5分钟。该值可以以“10s”或“5m”的形式指定（即分别为10秒或5分钟）。在所提供的最大等待时间上添加一小部分额外的等待时间，以分散任何并发请求的唤醒时间。这相当于等待/最多持续时间的16个额外时间。

### Implementation Details 实现细节

虽然该机制使用起来相对简单，但仍有一些必须正确处理的边缘情况。

* 如果索引倒退，请重置索引。虽然索引一般是单调递增的（即它们应该随着时间的推移而不断增加），但是有几种真实场景可以在给定查询的情况下向后转换。实现必须检查返回的索引是否低于先前的值，如果是，则应将索引重置为0 - 有效地重新启动其阻塞循环。如果不这样做可能会导致客户端在无限时间内错过未来的更新，或者使用无效的索引值导致无阻塞并增加服务器的负载。可能发生这种情况的情况包括：
  + 如果在具有较旧版本数据的服务器上恢复了raft快照。
  + KV列表操作，其中删除了具有最高索引的项目。
  + Consul升级改变了手表的工作方式，通过更细粒度的索引对其进行优化。
* 完整性检查指数大于零。在初始请求（或上面的重置）之后，返回的X-Consul-Index应始终大于零。如果不是Consul中的一个错误，但是这已经发生了几次，并且仍然可以在一些较旧的Consul版本上触发。它特别糟糕，因为它会导致阻塞的客户端无法进入繁忙的循环，使用过多的客户端CPU并导致服务器上的高负载。当请求的数据尚不存在时，使用索引1来等待更新总是安全的，因此客户端应该在处理每个阻塞响应之后检查它们的索引是否至少为1，以确保它们实际阻塞下一个请求。
* 速率限制。当更新相对较少时（更新之间的数十秒到几分钟的顺序），阻塞查询机制相当有效。如果结果得到更快的更新 - 可能是在中断或与行为不端的客户端发生事件时 - 阻塞查询循环会降级为占用过多客户端CPU并导致高服务器负载的繁忙循环。虽然可以在循环的每次迭代中添加一个睡眠，但不建议这样做，因为它会导致更新传递在快乐的情况下被延迟，并且它可能会加剧问题，因为它增加了索引在下一个请求。客户应该对循环进行速率限制，以便在快乐的情况下，它们无需等待即可继续进行，但是当值开始快速流失时，它们会以合理的速率（例如每15秒）降级为轮询。理想情况下，这是通过一种算法来完成的，该算法允许在开始限制速率之前进行几次快速连续交付 - 突发2的令牌桶是实现此目的的简单方法。

### Hash-based Blocking Queries

有限数量的代理端点也支持阻塞，但是因为状态是代理的本地状态而不是使用一致的raft索引来管理，所以它们的阻塞机制是不同的。

由于没有单调递增的索引，因此每个响应都包含一个标头X-Consul-ContentHash，它是通过对响应中相关的所有字段进行散列而生成的不透明散列摘要。

后续请求可以与查询参数hash = <value>一起发送，其中value是看到的最后一个哈希头值，并且这将阻塞直到等待超时被传递或者直到本地代理的状态以哈希将是的方式改变为止不同。

除了不同的头和查询参数名称之外，最大的区别是哈希值是不透明的，无法比较以查看一个结果是旧的还是比另一个更新。通常基于散列的阻塞由于幂等更新而不会过早返回，因为除非结果实际发生更改，否则散列将保持不变，但是与基于索引的阻塞一样，没有严格保证客户端永远不会观察到相同的结果在完全超时之前。

## Filtering

过滤器表达式用于优化某些API列表端点的数据查询，如各个API文档中所述。 在返回数据之前，将在Consul服务器上执行过滤，从而减少网络负载。 要使用数据查询将过滤器表达式传递给Consul，请使用filter参数。

curl -G <path> --data-urlencode 'filter=<filter expression>'

要创建过滤器表达式，您将使用匹配的运算符，选择器和值编写一个或多个表达式。

### Expression Syntax 表达式语法

表达式以纯文本格式书写。 支持布尔逻辑和括号。 通常，忽略空格，除了文字字符串。

#### Expressions

有几种连接表达式的方法，包括

* ogical or
* logical and
* logical not
* grouping with parenthesis
* matching expressions

// Logical Or - evaluates to true if either sub-expression does

<Expression 1> or <Expression 2>

// Logical And - evaluates to true if both sub-expressions do

<Expression 1 > and <Expression 2>

// Logical Not - evaluates to true if the sub-expression does not

not <Expression 1>

// Grouping - Overrides normal precedence rules

( <Expression 1> )

// Inspects data to check for a match

<Matching Expression 1>

各种形式可以预期标准运算符优先级。 例如，以下两个表达式是等效的。

<Expression 1> and not <Expression 2> or <Expression 3>

( <Expression 1> and (not <Expression 2> )) or <Expression 3>

#### Matching Operators

匹配运算符用于创建表达式。 所有匹配的运算符都使用选择器或值来选择应匹配的数据。 支持过滤的每个端点都接受可能不同的选择器列表，并在这些端点的API文档中进行了详细说明。

// Equality & Inequality checks

<Selector> == <Value>

<Selector> != <Value>

// Emptiness checks

<Selector> is empty

<Selector> is not empty

// Contains checks or Substring Matching

<Value> in <Selector>

<Value> not in <Selector>

<Selector> contains <Value>

<Selector> not contains <Value>

// Regular Expression Matching

<Selector> matches <Value>

<Selector> not matches <Value>

#### Selectors

匹配运算符使用选择器来创建表达式。 它们由a定义。 分隔的名单。 每个名称必须以ASCII字母开头，并且可以包含ASCII字母，数字和下划线。 当选择器的一部分引用地图值时，可以使用[“<map key name>”]形式表示，而不是。<map key name>。 这允许使用不是有效选择器的映射键的可能性。

// selects the foo key within the ServiceMeta mapping for the

// /catalog/service/:service endpoint

ServiceMeta.foo

// Also selects the foo key for the same endpoint

ServiceMeta["foo"]

#### Values

匹配运算符使用值来创建表达式。 值可以是任何有效的选择器，数字或带引号的字符串。 对于数字，任何基数为10的整数和浮点数都是可能的。 对于引用的字符串，它们可以用双引号或反引号括起来。 当用反引号括起来时，它们被视为原始字符串，而\ n的转义序列将不会被展开。

### Filter Utilization

通常，仅过滤主对象。 过滤不在顶层的数组中的项目时，将返回包含该项目的整个数组。 这通常是响应的最外层对象，但在某些情况下，例如/ catalog / node /：node端点，对结果中嵌入的对象执行过滤。

#### Performance

过滤器在服务器上执行，因此会占用服务器上的一些CPU时间。 对于非陈旧查询，这意味着过滤器在领导者上执行。

#### Filtering Examples

##### Agent API

**Command – Unfiltered**

curl -X GET localhost:8500/v1/agent/services

**Response - Unfiltered**

{

"redis1": {

"ID": "redis1",

"Service": "redis",

"Tags": [

"primary",

"production"

],

"Meta": {

"env": "production",

"foo": "bar"

},

"Port": 1234,

"Address": "",

"Weights": {

"Passing": 1,

"Warning": 1

},

"EnableTagOverride": **false**

},

"redis2": {

"ID": "redis2",

"Service": "redis",

"Tags": [

"secondary",

"production"

],

"Meta": {

"env": "production",

"foo": "bar"

},

"Port": 1235,

"Address": "",

"Weights": {

"Passing": 1,

"Warning": 1

},

"EnableTagOverride": **false**

},

"redis3": {

"ID": "redis3",

"Service": "redis",

"Tags": [

"primary",

"qa"

],

"Meta": {

"env": "qa"

},

"Port": 1234,

"Address": "",

"Weights": {

"Passing": 1,

"Warning": 1

},

"EnableTagOverride": **false**

}

}

**Command - Filtered**

curl -G localhost:8500/v1/agent/services --data-urlencode 'filter=Meta.env == qa'

**Response - Filtered**

{

"redis3": {

"ID": "redis3",

"Service": "redis",

"Tags": [

"primary",

"qa"

],

"Meta": {

"env": "qa"

},

"Port": 1234,

"Address": "",

"Weights": {

"Passing": 1,

"Warning": 1

},

"EnableTagOverride": **false**

}

}

##### Catalog API

**Command - Unfiltered**

curl -X GET localhost:8500/v1/catalog/service/api-internal

**Response - Unfiltered**

[

{

"ID": "b4f64e8c-5c7d-11e9-bf68-8c8590bd0966",

"Node": "node-1",

"Address": "198.18.0.1",

"Datacenter": "dc1",

"TaggedAddresses": **null**,

"NodeMeta": {

"agent": "true",

"arch": "i386",

"os": "darwin"

},

"ServiceKind": "",

"ServiceID": "api-internal",

"ServiceName": "api-internal",

"ServiceTags": [

"tag"

],

"ServiceAddress": "",

"ServiceWeights": {

"Passing": 1,

"Warning": 1

},

"ServiceMeta": {

"environment": "qa"

},

"ServicePort": 9090,

"ServiceEnableTagOverride": **false**,

"ServiceProxy": {},

"ServiceConnect": {},

"CreateIndex": 30,

"ModifyIndex": 30

},

{

"ID": "b4faf93a-5c7d-11e9-840d-8c8590bd0966",

"Node": "node-2",

"Address": "198.18.0.2",

"Datacenter": "dc1",

"TaggedAddresses": **null**,

"NodeMeta": {

"arch": "arm",

"os": "linux"

},

"ServiceKind": "",

"ServiceID": "api-internal",

"ServiceName": "api-internal",

"ServiceTags": [

"test",

"tag"

],

"ServiceAddress": "",

"ServiceWeights": {

"Passing": 1,

"Warning": 1

},

"ServiceMeta": {

"environment": "production"

},

"ServicePort": 9090,

"ServiceEnableTagOverride": **false**,

"ServiceProxy": {},

"ServiceConnect": {},

"CreateIndex": 29,

"ModifyIndex": 29

},

{

"ID": "b4fbe7f4-5c7d-11e9-ac82-8c8590bd0966",

"Node": "node-4",

"Address": "198.18.0.4",

"Datacenter": "dc1",

"TaggedAddresses": **null**,

"NodeMeta": {

"arch": "i386",

"os": "freebsd"

},

"ServiceKind": "",

"ServiceID": "api-internal",

"ServiceName": "api-internal",

"ServiceTags": [],

"ServiceAddress": "",

"ServiceWeights": {

"Passing": 1,

"Warning": 1

},

"ServiceMeta": {

"environment": "qa"

},

"ServicePort": 9090,

"ServiceEnableTagOverride": **false**,

"ServiceProxy": {},

"ServiceConnect": {},

"CreateIndex": 28,

"ModifyIndex": 28

}

]

**Command - Filtered**

curl -G localhost:8500/v1/catalog/service/api-internal --data-urlencode 'filter=NodeMeta.os == linux'

**Response - Filtered**

[

{

"ID": "b4faf93a-5c7d-11e9-840d-8c8590bd0966",

"Node": "node-2",

"Address": "198.18.0.2",

"Datacenter": "dc1",

"TaggedAddresses": **null**,

"NodeMeta": {

"arch": "arm",

"os": "linux"

},

"ServiceKind": "",

"ServiceID": "api-internal",

"ServiceName": "api-internal",

"ServiceTags": [

"test",

"tag"

],

"ServiceAddress": "",

"ServiceWeights": {

"Passing": 1,

"Warning": 1

},

"ServiceMeta": {

"environment": "production"

},

"ServicePort": 9090,

"ServiceEnableTagOverride": **false**,

"ServiceProxy": {},

"ServiceConnect": {},

"CreateIndex": 29,

"ModifyIndex": 29

}

]

##### Health API

**Command - Unfiltered**

curl -X GET localhost:8500/v1/health/node/node-1

**Response - Unfiltered**

[

{

"Node": "node-1",

"CheckID": "node-health",

"Name": "Node level check",

"Status": "critical",

"Notes": "",

"Output": "",

"ServiceID": "",

"ServiceName": "",

"ServiceTags": [],

"Definition": {},

"CreateIndex": 13,

"ModifyIndex": 13

},

{

"Node": "node-1",

"CheckID": "svc-web-health",

"Name": "Service level check - web",

"Status": "warning",

"Notes": "",

"Output": "",

"ServiceID": "",

"ServiceName": "web",

"ServiceTags": [],

"Definition": {},

"CreateIndex": 18,

"ModifyIndex": 18

}

]

**Command - Filtered**

curl -G localhost:8500/v1/health/node/node-1 --data-urlencode 'filter=ServiceName != ""'

**Response - Filtered**

[

{

"Node": "node-1",

"CheckID": "svc-web-health",

"Name": "Service level check - web",

"Status": "warning",

"Notes": "",

"Output": "",

"ServiceID": "",

"ServiceName": "web",

"ServiceTags": [],

"Definition": {},

"CreateIndex": 18,

"ModifyIndex": 18

}

]

## Agent Caching

一些读取端点支持代理缓存。它们在文件中有明确的标记。代理缓存可以采取两种形式，简单刷新或后台刷新，这取决于端点的语义。每个端点的文档清楚地标识支持哪种形式的缓存。详细情况如下。

在支持缓存的地方，可以通过 ?cached参数启动缓存。不能错误的 把?cached 和 ?consistent组合一起

### Simple Caching

支持简单缓存的端点可以直接从本地代理的缓存返回结果，而无需往返服务器。 默认情况下，代理会将结果缓存相对较长的时间（3天），这样即使服务器在较长时间内不可用以启用“失败静态”语义，它仍然可以返回结果。

这意味着在没有其他参数的情况下，缓存的查询可能会收到几天前的响应。为了要求更好的新鲜度，HTTP缓存-控制报头可以设置一个指令，比如max-age=<秒>。在这种情况下，如果缓存的值早于给定的最大年龄，代理将尝试从服务器重新获取结果。如果无法到达服务器，将正常返回500。

为了允许客户端在正常操作中保持新的结果，但是如果服务器不可用，则允许客户端使用过时的结果，可以在Cache-Control头中另外提供stale-if-error = <seconds>指令。 这将返回缓存的值，即使它比max-age更早（假设它不比stale-if-error更早）而不是500.它必须与max-age或must-revalidate一起提供。 如果大于max-age，则Age响应头可用于确定服务器是否无法访问，而是返回缓存版本。

例如，假设存在65秒的缓存响应，并且服务器当前不可用，则Cache-Control：max-age = 30将导致500错误，而Cache-Control：max-age = 30 stale -if-error = 259200将导致返回缓存的响应。

请求设置max-age = 0或must-revalidate指令将导致代理始终从服务器重新获取响应。 可以将两者结合使用stale-if-error = <seconds>以确保服务器可用时的新结果，但如果对服务器的请求失败，则返回到缓存结果。

不使用缓存的请求当前完全绕过缓存，因此返回的缓存响应可能比同一代理上返回的最后一个未缓存响应更陈旧。 如果这会导致问题，则可以使用？cached和设置Cache-Control：must-revalidate来生成请求，以获得始终保持新鲜的结果，同时保持缓存填充最新结果。

在所有情况下，HTTPX-Cache报头总是在响应中设置为命中或未命中，指示响应是否来自缓存。

对于缓存命中，总是在响应中设置HTTPAGE报头，以指示从服务器获取该响应后的秒数。

### Background Refresh Caching

支持后台刷新缓存的端点可以直接从本地代理的缓存返回结果，而无需往返服务器。作为未命中的第一次提取将导致从服务器进行初始提取，但还将触发代理开始后台阻止查询，以监视对该结果的任何更改，并在发生更改时更新缓存值。

在没有请求TTL资源（通常为3天）之前，以下请求将始终是缓存命中。

客户端可以对将从缓存提供的本地代理执行阻止查询。 这允许多个客户端在本地观察相同的资源，而对来自给定客户端代理的服务器只有一个阻止监视该资源。

在此模式下，将忽略HTTP Cache-Control标头，因为缓存正在主动更新，并且与典型的被动缓存具有不同的语义。

在所有情况下，HTTP X-Cache标头始终在对HIT或MISS的响应中设置，指示响应是否是从缓存提供的。

对于缓存命中，始终在响应中设置HTTP Age标头，以指示自从服务器获取响应以来的秒数。 只要本地代理与服务器具有活动连接，则年龄将始终为0，因为该值是最新的。 如果代理程序断开连接，则仍会返回缓存的结果，但使用Age表示自本地代理程序与服务器断开连接以来经过了多少秒，在此期间可能错过了对结果的更新。