

Table of Contents

MINIO	1.1
MINIO服务器	1.2
MinIO快速入门指南	1.2.1
MinIO Docker快速入门	1.2.2
MinIO纠删码快速入门	1.2.3
分布式MinIO快速入门	1.2.4
使用MinIO分解后的Spark和Hadoop Hive	1.2.5
使用TLS安全的访问MinIO服务	1.2.6
MinIO存储桶通知指南	1.2.7
MinIO服务限制/租户	1.2.8
MinIO Server配置指南	1.2.9
MinIO多租户(Multi-tenant)部署指南	1.2.10
MinIO Azure网关	1.2.11
MinIO GCS网关	1.2.12
MinIO NAS网关	1.2.13
MinIO S3网关	1.2.14
MinIO HDFS网关	1.2.15
MinIO磁盘缓存指南	1.2.16
MinIO监控指南	1.2.17
如何使用Prometheus监控MinIO	1.2.18
MinIO Federation快速入门指南	1.2.19
MinIO KMS快速入门指南	1.2.20
MinIO Select API快速入门指南	1.2.21
MinIO压缩指南	1.2.22
MinIO多用户快速入门指南	1.2.23
MinIO STS快速入门指南	1.2.24
MINIO部署	1.3
MinIO部署快速入门	1.3.1
使用Docker Swarm部署MinIO	1.3.2
使用Kubernetes部署MinIO	1.3.3
使用Docker Compose部署MinIO	1.3.4
MINIO客户端	1.4
MinIO客户端快速入门指南	1.4.1
MinIO Client完全指南	1.4.2
MinIO Admin完全指南	1.4.3
MINIO SDKS	1.5
JavaScript Client快速入门指南	1.5.1
JavaScript Client API参考文档	1.5.2
Java Client快速入门指南	1.5.3
Java Client API参考文档	1.5.4

Python Client快速入门指南	1.5.5
Python Client API参考文档	1.5.6
Golang Client快速入门指南	1.5.7
Golang Client API参考文档	1.5.8
.NET Client快速入门指南	1.5.9
.NET Client API参考文档	1.5.10
Haskell客户端快速入门指南	1.5.11
Haskell客户端API参考	1.5.12
实战秘籍	1.6
使用S3cmd操作MinIO	1.6.1
使用AWS CLI结合MinIO	1.6.2
restic结合MinIO	1.6.3
将MySQL备份存储到MinIO	1.6.4
将MongoDB备份存储到MinIO	1.6.5
将PostgreSQL备份存储到MinIO	1.6.6
为MinIO设置Caddy proxy	1.6.7
为MinIO设置Nginx代理	1.6.8
将Apache日志存储到MinIO	1.6.9
Rclone结合MinIO Server	1.6.10
结合MinIO运行Deis Workflow	1.6.11
为MinIO Server设置Apache HTTP proxy	1.6.12
使用pre-signed URLs通过浏览器上传	1.6.13
如何在FreeNAS中运行MinIO	1.6.14
如何使用Cyberduck结合MinIO	1.6.15
如何使用AWS SDK for PHP操作MinIO Server	1.6.16
如何使用AWS SDK for Ruby操作MinIO Server	1.6.17
如何使用AWS SDK for Python操作MinIO Server	1.6.18
如何使用Mountain Duck结合MinIO	1.6.19
如何使用AWS SDK for javascript操作MinIO Server	1.6.20
如何使用Træfik代理多个MinIO服务	1.6.21
如何使用AWS SDK for Go操作MinIO Server	1.6.22
如何使用AWS SDK for Java操作MinIO Server	1.6.23
如何使用Paperclip操作MinIO Server	1.6.24
如何使用AWS SDK for .NET操作MinIO Server	1.6.25
如何使用aws-cli调用MinIO服务端加密	1.6.26

MinIO 是在 **GNU Affero 通用公共许可证 v3.0** 下发布的高性能对象存储。它是与 Amazon S3 云存储服务兼容的 API。使用 MinIO 为机器学习、分析和应用程序数据工作负载构建高性能基础架构。

MinIO Quickstart Guide



MinIO 是在 GNU Affero 通用公共许可证 v3.0 下发布的高性能对象存储。它是与 Amazon S3 云存储服务兼容的 API。使用 MinIO 为机器学习、分析和应用程序数据工作负载构建高性能基础架构。

248 / 5000 翻译结果 此自述文件提供了在裸机硬件上运行 MinIO 的快速入门说明，包括基于容器的安装。对于 Kubernetes 环境，请使用 [MinIO Kubernetes Operator](#)。

容器安装

使用以下命令将独立的 MinIO 服务器作为容器运行。

独立的 MinIO 服务器最适合早期开发和评估。某些功能，例如版本控制、对象锁定和存储桶复制 需要使用擦除编码分布式部署 MinIO。对于扩展的开发和生产，请在启用擦除编码的情况下部署 MinIO - 特别是，每个 MinIO 服务器最少 4 个驱动器。详见【MinIO擦除码快速入门指南】(<http://docs.minio.org.cn/docs/minio-erasure-code-quickstart-guide.html>) 获取更完整的文档。

稳定

运行以下命令以使用临时数据卷将 MinIO 的最新稳定映像作为容器运行：

```
podman run \
-p 9000:9000 \
-p 9001:9001 \
minio/minio server /data --console-address ":9001"
```

MinIO 部署开始使用默认的 root 凭据 `minioadmin:minioadmin`。您可以使用 MinIO 控制台测试部署，这是一个嵌入式内置于 MinIO 服务器的对象浏览器。将主机上运行的 Web 浏览器指向 <http://127.0.0.1:9000> 并使用 根凭据。您可以使用浏览器来创建桶、上传对象以及浏览 MinIO 服务器的内容。

您还可以使用任何与 S3 兼容的工具进行连接，例如 MinIO Client `mc` 命令行工具。见 [使用 MinIO 客户端 `mc` 进行测试](#) 了解有关使用 `mc` 命令行工具的更多信息。对于应用程序开发人员，请参阅 <http://docs.minio.org.cn/docs/> 并单击导航中的 **MinIO SDKs** 以查看支持语言的 MinIO SDK。

注意：要在持久存储上部署 MinIO，您必须使用 `podman -v` 选项将本地持久目录从主机操作系统映射到容器。例如，`-v /mnt/data:/data` 将位于 `/mnt/data` 的主机操作系统驱动器映射到容器上的 `/data`。

macOS

使用以下命令在 macOS 上运行独立的 MinIO 服务器。

独立的 MinIO 服务器最适合早期开发和评估。某些功能（例如版本控制、对象锁定和存储桶复制）需要使用擦除编码分布式部署 MinIO。对于扩展的开发和生产，部署启用擦除编码的 MinIO - 具体来说，每个 MinIO 服务器最少 4 个驱动器。更完整的文档请参见 [【MinIO 擦除码快速入门指南】](#) (<http://docs.minio.org.cn/docs/minio-erasure-code-quickstart-guide.html>)。

Homebrew (推荐)

运行以下命令以使用 Homebrew 安装最新的稳定 MinIO 包。将 `/data` 替换为您希望 MinIO 存储数据的驱动器或目录的路径。

```
brew install minio/stable/minio
minio server /data
```

注意：如果你之前使用 `brew install minio` 安装过 minio，那么建议你从 `minio/stable/minio` 官方 repo 重新安装 minio。

```
brew uninstall minio
brew install minio/stable/minio
```

MinIO 部署开始使用默认的 root 凭据 `minioadmin:minioadmin`。您可以使用 MinIO 控制台测试部署，这是一个内置在 MinIO 服务器中的基于 Web 的嵌入式对象浏览器。将主机上运行的 Web 浏览器指向 <http://127.0.0.1:9000> 并使用 root 凭据登录。您可以使用浏览器来创建桶、上传对象以及浏览 MinIO 服务器的内容。

您还可以使用任何与 S3 兼容的工具进行连接，例如 MinIO Client `mc` 命令行工具。有关使用 `mc` 命令行工具的更多信息，请参阅 [使用 MinIO 客户端 `mc` 进行测试](#)。对于应用程序开发人员，请参阅 <http://docs.minio.org.cn/docs/> 并单击导航中的 **MinIO SDKs** 以查看支持语言的 MinIO SDK。

二进制下载

使用以下命令在 macOS 上下载并运行独立的 MinIO 服务器。将 `/data` 替换为您希望 MinIO 存储数据的驱动器或目录的路径。

```
wget http://dl.minio.org.cn/server/minio/release/darwin-amd64/minio
chmod +x minio
./minio server /data
```

MinIO 部署开始使用默认的 root 凭据 `minioadmin:minioadmin`。您可以使用 MinIO 控制台测试部署，这是一个内置在 MinIO 服务器中的基于 Web 的嵌入式对象浏览器。将主机上运行的 Web 浏览器指向 <http://127.0.0.1:9000> 并使用 root 凭据登录。您可以使用浏览器来创建桶、上传对象以及浏览 MinIO 服务器的内容。

您还可以使用任何与 S3 兼容的工具进行连接，例如 MinIO Client `mc` 命令行工具。有关使用 `mc` 命令行工具的更多信息，请参阅 [使用 MinIO 客户端 `mc` 进行测试](#)。对于应用程序开发人员，请参阅 <http://docs.minio.org.cn/docs/> 并单击导航中的 **MinIO SDKs** 以查看支持语言的 MinIO SDK。

GNU/Linux

使用以下命令在运行 64 位 Intel/AMD 架构的 Linux 主机上运行独立的 MinIO 服务器。将 `/data` 替换为您希望 MinIO 存储数据的驱动器或目录的路径。

```
wget http://dl.minio.org.cn/server/minio/release/linux-amd64/minio
chmod +x minio
./minio server /data
```

将 `/data` 替换为您希望 MinIO 存储数据的驱动器或目录的路径。

下表列出了支持的架构。将 `wget URL` 替换为您的 Linux 主机的架构。

Architecture	URL
64-bit Intel/AMD	http://dl.minio.org.cn/server/minio/release/linux-amd64/minio
64-bit ARM	http://dl.minio.org.cn/server/minio/release/linux-arm64/minio
64-bit PowerPC LE (ppc64le)	http://dl.minio.org.cn/server/minio/release/linux-ppc64le/minio
IBM Z-Series (S390X)	http://dl.minio.org.cn/server/minio/release/linux-s390x/minio

MinIO 部署开始使用默认的 root 凭据 `minioadmin:minioadmin`。您可以使用 MinIO 控制台测试部署，这是一个内置在 MinIO 服务器中的基于 Web 的嵌入式对象浏览器。将主机上运行的 Web 浏览器指向 <http://127.0.0.1:9000> 并使用 root 凭据登录。您可以使用浏览器来创建桶、上传对象以及浏览 MinIO 服务器的内容。

您还可以使用任何与 S3 兼容的工具进行连接，例如 MinIO Client `mc` 命令行工具。有关使用 `mc` 命令行工具的更多信息，请参阅 [使用 MinIO 客户端 mc 进行测试](#)。对于应用程序开发人员，请参阅 <http://docs.minio.org.cn/docs/> 并单击导航中的 **MinIO SDKs** 以查看支持语言的 MinIO SDK。

注意：独立的 MinIO 服务器最适合早期开发和评估。某些功能（例如版本控制、对象锁定和存储桶复制）需要使用擦除编码分布式部署 MinIO。对于扩展的开发和生产，部署启用擦除编码的 MinIO - 具体来说，每个 MinIO 服务器最少 4 个驱动器。更完整的文档请参见【MinIO 擦除码快速入门指南】(<http://docs.minio.org.cn/docs/minio-erasure-code-quickstart-guide.html>)。

微软视窗

要在 64 位 Windows 主机上运行 MinIO，请从以下 URL 下载 MinIO 可执行文件：

```
http://dl.minio.org.cn/server/minio/release/windows-amd64/minio.exe
```

使用以下命令在 Windows 主机上运行独立的 MinIO 服务器。将“D:\”替换为您希望 MinIO 存储数据的驱动器或目录的路径。您必须将终端或 powershell 目录更改为 `minio.exe` 可执行文件的位置，或将该目录的路径添加到系统 `$PATH` 中：

```
minio.exe server D:\
```

MinIO 部署开始使用默认的 root 凭据 `minioadmin:minioadmin`。您可以使用 MinIO 控制台测试部署，这是一个内置在 MinIO 服务器中的基于 Web 的嵌入式对象浏览器。将主机上运行的 Web 浏览器指向 <http://127.0.0.1:9000> 并使用 root 凭据登录。您可以使用浏览器来创建桶、上传对象以及浏览 MinIO 服务器的内容。

您还可以使用任何与 S3 兼容的工具进行连接，例如 MinIO Client `mc` 命令行工具。有关使用 `mc` 命令行工具的更多信息，请参阅 [使用 MinIO 客户端 mc 进行测试](#)。对于应用程序开发人员，请参阅 <http://docs.minio.org.cn/docs/> 并单击导航中的 **MinIO SDKs** 以查看支持语言的 MinIO SDK。

注意：独立的 MinIO 服务器最适合早期开发和评估。某些功能（例如版本控制、对象锁定和存储桶复制）需要使用擦除编码分布式部署 MinIO。对于扩展的开发和生产，部署启用擦除编码的 MinIO - 具体来说，每个 MinIO 服务器最少 4 个驱动器。更完整的文档请参见【MinIO 擦除码快速入门指南】(<http://docs.minio.org.cn/docs/minio-erasure-code-quickstart-guide.html>)。

[quickstart-guide.html](#))。

从源码安装

使用以下命令从源代码编译和运行独立的 MinIO 服务器。源代码安装仅适用于开发人员和高级用户。如果您没有可用的 Golang 环境, 请参考【如何安装 Golang】(<https://golang.org/doc/install>)。所需的最低版本是 go1.16

```
GO111MODULE=on go install github.com/minio/minio@latest
```

MinIO 部署开始使用默认的 root 凭据 `minioadmin:minioadmin`。您可以使用 MinIO 控制台测试部署, 这是一个内置在 MinIO 服务器中的基于 Web 的嵌入式对象浏览器。将主机上运行的 Web 浏览器指向 <http://127.0.0.1:9000> 并使用 root 凭据登录。您可以使用浏览器来创建桶、上传对象以及浏览 MinIO 服务器的内容。

您还可以使用任何与 S3 兼容的工具进行连接, 例如 MinIO Client `mc` 命令行工具。有关使用 `mc` 命令行工具的更多信息, 请参阅 [使用 MinIO 客户端 mc 进行测试](#)。对于应用程序开发人员, 请参阅 <http://docs.minio.org.cn/docs/> 并单击导航中的 **MinIO SDKs** 以查看支持语言的 MinIO SDK。

注意: 独立的 MinIO 服务器最适合早期开发和评估。某些功能(例如版本控制、对象锁定和存储桶复制)需要使用擦除编码分布式部署 MinIO。对于扩展的开发和生产, 部署启用擦除编码的 MinIO - 具体来说, 每个 MinIO 服务器最少 4 个驱动器。更完整的文档请参见【MinIO 擦除码快速入门指南】(<http://docs.minio.org.cn/docs/minio-erasure-code-quickstart-guide.html>)。

MinIO 强烈建议反对 在生产环境中使用从源代码编译的 MinIO 服务器。

部署建议

允许防火墙的端口访问

默认情况下, MinIO 使用端口 9000 来侦听传入连接。如果您的平台默认阻止该端口, 您可能需要启用对该端口的访问。

ufw

对于启用了 ufw 的主机(基于 Debian 的发行版), 您可以使用 `ufw` 命令来允许流量到特定端口。使用以下命令允许访问端口 9000

```
ufw allow 9000
```

下面的命令启用所有传入端口的流量, 范围从 9000 到 9010。

```
ufw allow 9000:9010/tcp
```

防火墙-cmd

对于启用了 `firewall-cmd`(CentOS) 的主机, 您可以使用 `firewall-cmd` 命令来允许特定端口的流量。使用以下命令允许访问端口 9000

```
firewall-cmd --get-active-zones
```

此命令获取活动区域。现在, 将端口规则应用于上面返回的相关区域。例如, 如果区域是 `public`, 请使用

```
firewall-cmd --zone=public --add-port=9000/tcp --permanent
```

请注意，“permanent”确保规则在防火墙启动、重启或重新加载时是持久的。最后重新加载防火墙以使更改生效。

```
firewall-cmd --reload
```

iptables

对于启用了 `iptables` 的主机（RHEL、CentOS 等），您可以使用 `iptables` 命令来启用进入特定端口的所有流量。使用以下命令允许访问 9000 端口

```
iptables -A INPUT -p tcp --dport 9000 -j ACCEPT  
service iptables restart
```

下面的命令启用所有传入端口的流量，范围从 9000 到 9010。

```
iptables -A INPUT -p tcp --dport 9000:9010 -j ACCEPT  
service iptables restart
```

预先存在的数据

当部署在单个驱动器上时，MinIO 服务器允许客户端访问数据目录中的任何预先存在的数据。例如，如果 MinIO 是用命令 `minio server /mnt/data` 启动的，`/mnt/data` 目录中的任何预先存在的数据都可以被客户端访问。

上述语句也适用于所有网关后端。

测试 MinIO 连接

使用 MinIO 控制台进行测试

MinIO Server 带有一个基于 Web 的嵌入式对象浏览器。将您的 Web 浏览器指向 <http://127.0.0.1:9000> 以确保您的服务器已成功启动。

注意：默认情况下，MinIO 在随机端口上运行控制台，如果您希望选择特定端口，请使用 `--console-address` 来选择特定接口和端口。

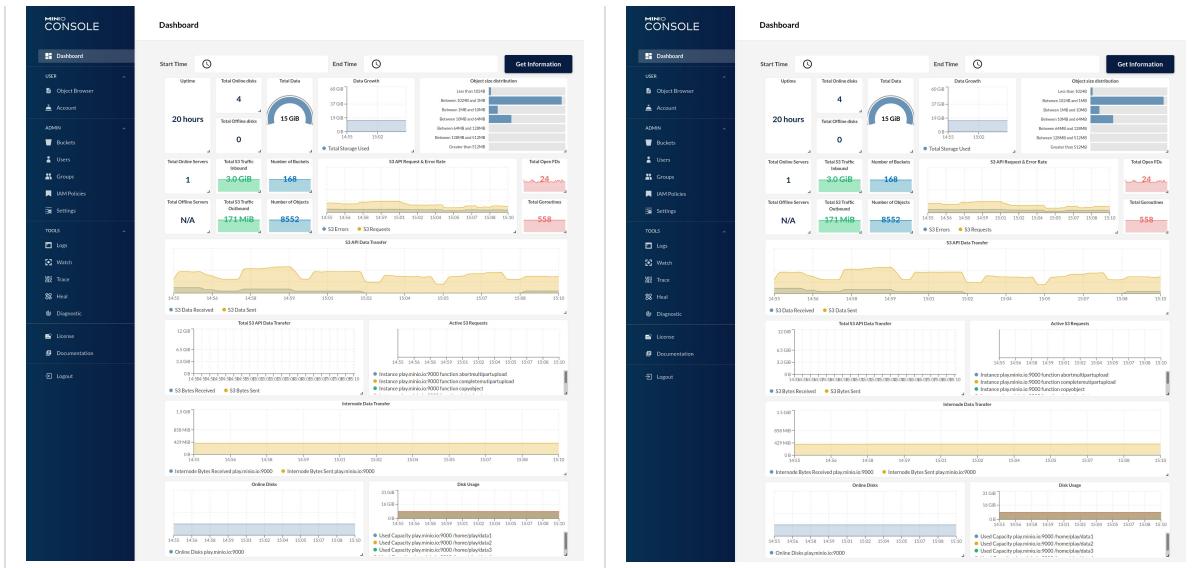
需要考虑的事情

MinIO 将浏览器访问请求重定向到配置的服务器端口（即 `127.0.0.1:9000`）到配置的控制台端口。MinIO 在构建重定向 URL 时使用请求中指定的主机名或 IP 地址。URL 和端口必须可由客户端访问才能使重定向工作。

对于 MinIO 主机 IP 地址或端口不公开的负载均衡器、代理或入口规则后面的部署，请使用 `MINIO_BROWSER_REDIRECT_URL` 环境变量为重定向指定外部主机名。LB/Proxy 必须有专门将流量定向到控制台端口的规则。

例如，考虑在代理 `https://minio.example.net`、`https://console.minio.example.net` 后面的 MinIO 部署，其规则将端口 :9000 和 :9001 上的流量转发到 MinIO 和分别在内部网络上的 MinIO Console。将 `MINIO_BROWSER_REDIRECT_URL` 设置为 `https://console.minio.example.net` 以确保浏览器接收到有效的可访问 URL。

Dashboard	Creating a bucket



使用 `MinIO Client mc` 进行测试

`mc` 为 `ls`、`cat`、`cp`、`mirror`、`diff` 等 UNIX 命令提供了一种现代替代方案。它支持文件系统和 Amazon S3 兼容的云存储服务。按照 MinIO 客户端 [快速入门指南](#) 获取更多说明。

升级 MinIO

MinIO 服务器支持滚动升级，即您可以在分布式集群中一次更新一个 MinIO 实例。这允许在不停机的情况下进行升级。升级可以通过用最新版本替换二进制文件并以滚动方式重新启动所有服务器来手动完成。但是，我们建议所有用户从客户端使用 `mc admin update`。这将同时更新集群中的所有节点并重新启动它们，如来自 MinIO 客户端 (`mc`) 的以下命令所示：

```
mc admin update <minio alias, e.g., myminio>
```

注意：某些版本可能不允许滚动升级，这总是在发行说明中提到，通常建议在升级前阅读发行说明。在这种情况下，`mc admin update` 是一次升级所有服务器的推荐升级机制。

MinIO 升级期间要记住的重要事项

- `mc admin update` 仅在运行 MinIO 的用户对二进制文件所在的父目录具有写访问权限时才有效，例如，如果当前二进制文件位于 `/usr/local/bin/minio`，则需要写入访问 `/usr/local/bin`。
- `mc admin update` 同时更新和重启所有服务器，应用程序会在升级后重试并继续各自的操作。
- `mc admin update` 在 kubernetes/container 环境中被禁用，容器环境提供自己的机制来推出更新。
- 在联合设置的情况下，`mc admin update` 应该单独针对每个集群运行。在所有集群成功更新之前，避免将 `mc` 更新为任何新版本。
- 如果使用 `kes` 作为 MinIO 的 KMS，只需替换二进制文件并重新启动 `kes` 有关 `kes` 的更多信息可以在 [here](#) 中找到
- 如果将 `Vault` 作为 KMS 与 MinIO 一起使用，请确保您已遵循此处概述的 `Vault` 升级过程：<https://www.vaultproject.io/docs/upgrading/index.html>
- 如果将 `etcd` 与 MinIO 用于联合，请确保您已遵循此处概述的 `etcd` 升级过程：<https://github.com/etcd-io/etcd/blob/master/Documentation/upgrades/upgrading-etcd.md>

进一步探索

- [MinIO 擦除码快速入门指南](#)

- 在 MinIO 服务器上使用 `mc`
- 在 MinIO 服务器上使用 `aws-cli`
- 在 MinIO 服务器上使用 `s3cmd`
- 在 MinIO 服务器上使用 `minio-go` SDK
- [MinIO 文档网站](#)

为 **MinIO** 项目做贡献

请遵循 [MinIO 贡献者指南](#)

许可证

MinIO 的使用受 [GNU AGPLv3](#) 许可证管理，该许可证可在 [许可证](#) 文件中找到。

MinIO Docker 快速入门

前提条件

您的机器已经安装docker。从[这里](#)下载相关软件。

在Docker中运行MinIO单点模式。

MinIO 需要一个持久卷来存储配置和应用数据。不过, 如果只是为了测试一下, 您可以通过简单地传递一个目录(在下面的示例中为`/data`)启动MinIO。这个目录会在容器启动时在容器的文件系统中创建, 不过所有的数据都会在容器退出时丢失。

```
docker run -p 9000:9000 minio/minio server /data
```

要创建具有永久存储的MinIO容器, 您需要将本地持久目录从主机操作系统映射到虚拟配置`~/.minio` 并导出`/data` 目录。为此, 请运行以下命令

GNU/Linux 和 macOS

```
docker run -p 9000:9000 --name minio1 \
-v /mnt/data:/data \
-v /mnt/config:/root/.minio \
minio/minio server /data
```

Windows

```
docker run -p 9000:9000 --name minio1 \
-v D:\data:/data \
-v D:\minio\config:/root/.minio \
minio/minio server /data
```

在Docker中运行MinIO分布式模式

分布式MinIO可以通过[Docker Compose](#) 或者[Swarm mode](#)进行部署。这两者之间的主要区别是Docker Compose创建了单个主机, 多容器部署, 而Swarm模式创建了一个多主机, 多容器部署。

这意味着Docker Compose可以让你快速的在你的机器上快速使用分布式MinIO-非常适合开发, 测试环境; 而Swarm模式提供了更健壮, 生产级别的部署。

MinIO Docker提示

MinIO自定义Access和Secret密钥

要覆盖MinIO的自动生成的密钥, 您可以将Access和Secret密钥设为环境变量。MinIO允许常规字符串作为Access和Secret密钥。

GNU/Linux 和 macOS

```
docker run -p 9000:9000 --name minio1 \
-e "MINIO_ACCESS_KEY=AKIAIOSFODNN7EXAMPLE" \
-e "MINIO_SECRET_KEY=wJaIrxUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY" \
```

```
-v /mnt/data:/data \
-v /mnt/config:/root/.minio \
minio/minio server /data
```

Windows

```
docker run -p 9000:9000 --name minio1 \
-e "MINIO_ACCESS_KEY=AKIAIOSFODNN7EXAMPLE" \
-e "MINIO_SECRET_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY" \
-v D:\data:/data \
-v D:\minio\config:/root/.minio \
minio/minio server /data
```

使用Docker secrets进行MinIO Access和Secret密钥自定义

要覆盖MinIO的自动生成的密钥,你可以把secret和access秘钥创建成Docker secrets. MinIO允许常规字符串作为Access和Secret密钥。

```
echo "AKIAIOSFODNN7EXAMPLE" | docker secret create access_key -
echo "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY" | docker secret create secret_key -
```

使用 `docker service` 创建MinIO服务, 并读取Docker secrets。

```
docker service create --name="minio-service" --secret="access_key" --secret="secret_key" minio/minio server /data
```

更多 `docker service` 信息, 请访问 [这里](#)

获取容器ID

在容器中使用Docker命令, 你需要知道这个容器的 容器ID 。为了获取 Container ID ,运行

```
docker ps -a
```

`-a` flag 确保你获取所有的容器(创建的, 正在运行的, 退出的), 然后从输出中识别 Container ID 。

启动和停止容器

启动容器,你可以使用 `docker start` 命令。

```
docker start <container_id>
```

停止一下正在运行的容器, 使用 `docker stop` 命令。

```
docker stop <container_id>
```

MinIO容器日志

获取MinIO日志, 使用 `docker logs` 命令。

```
docker logs <container_id>
```

监控**MinioDocker**容器

监控MinIO容器使用的资源,使用 `docker stats` 命令.

```
docker stats <container_id>
```

了解更多

- [在Docker Compose上部署MinIO](#)
- [在Docker Swarm上部署MinIO](#)
- [分布式MinIO快速入门](#)
- [MinIO纠删码模式快速入门](#)

Minio纠删码快速入门

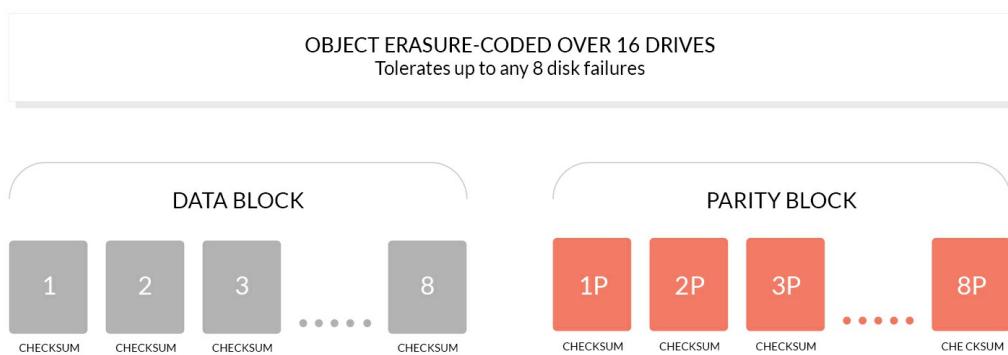
Minio使用纠删码 **erasure code** 和校验和 **checksum** 来保护数据免受硬件故障和无声数据损坏。即便您丢失一半数量 ($N/2$) 的硬盘，您仍然可以恢复数据。

什么是纠删码 **erasure code** ?

纠删码是一种恢复丢失和损坏数据的数学算法，Minio采用Reed-Solomon code将对象拆分成 $N/2$ 数据块和 $N/2$ 奇偶校验块。这就意味着如果是12块盘，一个对象会被分成6个数据块、6个奇偶校验块，你可以丢失任意6块盘（不管其是存放的数据块还是奇偶校验块），你仍可以从剩下的盘中的数据进行恢复，是不是很NB，感兴趣的同学请翻墙google。

为什么纠删码有用？

纠删码的工作原理和RAID或者复制不同，像RAID6可以在损失两块盘的情况下不丢数据，而Minio纠删码可以在丢失一半的盘的情况下，仍可以保证数据安全。而且Minio纠删码是作用在对象级别，可以一次恢复一个对象，而RAID是作用在卷级别，数据恢复时间很长。Minio对每个对象单独编码，存储服务一经部署，通常情况下是不需要更换硬盘或者修复。Minio纠删码的设计目标是为了性能和尽可能的使用硬件加速。



什么是位衰减 **bit rot** 保护？

位衰减又被称为数据腐化 **Data Rot**、无声数据损坏 **Silent Data Corruption**，是目前硬盘数据的一种严重数据丢失问题。硬盘上的数据可能会神不知鬼不觉就损坏了，也没有什么错误日志。正所谓明枪易躲，暗箭难防，这种背地里犯的错比硬盘直接咔咔宕了还危险。不过不用怕，Minio纠删码采用了高速 **HighwayHash** 基于哈希的校验和来防范位衰减。

Minio纠删码快速入门

1. 前提条件：

安装Minio- [Minio快速入门](#)

2. 以纠删码模式运行Minio

示例：使用Minio，在12个盘中启动Minio服务。

```
minio server /data1 /data2 /data3 /data4 /data5 /data6 /data7 /data8 /data9 /data10 /data11 /data12
```

示例: 使用Minio Docker镜像, 在8块盘中启动Minio服务。

```
docker run -p 9000:9000 --name minio \
-v /mnt/data1:/data1 \
-v /mnt/data2:/data2 \
-v /mnt/data3:/data3 \
-v /mnt/data4:/data4 \
-v /mnt/data5:/data5 \
-v /mnt/data6:/data6 \
-v /mnt/data7:/data7 \
-v /mnt/data8:/data8 \
minio/minio server /data1 /data2 /data3 /data4 /data5 /data6 /data7 /data8
```

3. 验证是否设置成功

你可以随意拔掉硬盘, 看Minio是否可以正常读写。

分布式MinIO快速入门

分布式Minio可以让你将多块硬盘（甚至在不同的机器上）组成一个对象存储服务。由于硬盘分布在不同的节点上，分布式Minio避免了单点故障。

分布式Minio有什么好处？

在大数据领域，通常的设计理念都是无中心和分布式。Minio分布式模式可以帮助你搭建一个高可用的对象存储服务，你可以使用这些存储设备，而不用考虑其真实物理位置。

数据保护

分布式Minio采用[纠删码](#)来防范多个节点宕机和[位衰减 bit rot](#)。

分布式Minio至少需要4个硬盘，使用分布式Minio自动引入了纠删码功能。

高可用

单机Minio服务存在单点故障，相反，如果是一个有N块硬盘的分布式Minio，只要有N/2硬盘在线，你的数据就是安全的。不过你需要至少有N/2+1个硬盘来创建新的对象。

例如，一个16节点的Minio集群，每个节点16块硬盘，就算8台服务器宕机，这个集群仍然是可读的，不过你需要9台服务器才能写数据。

注意，只要遵守分布式Minio的限制，你可以组合不同的节点和每个节点几块硬盘。比如，你可以使用2个节点，每个节点4块硬盘，也可以使用4个节点，每个节点两块硬盘，诸如此类。

一致性

Minio在分布式和单机模式下，所有读写操作都严格遵守[read-after-write](#)一致性模型。

开始吧

如果你了解Minio单机模式的搭建的话，分布式搭建的流程基本一样，Minio服务基于命令行传入的参数自动切换成单机模式还是分布式模式。

1. 前提条件

安装Minio - [Minio快速入门](#).

2. 运行分布式Minio

启动一个分布式Minio实例，你只需要把硬盘位置做为参数传给minio server命令即可，然后，你需要在所有其它节点运行同样的命令。

注意

- 分布式Minio里所有的节点需要有同样的access秘钥和secret秘钥，这样这些节点才能建立联接。为了实现这个，你需要在执行minio server命令之前，先将access秘钥和secret秘钥export成环境变量。
- 分布式Minio使用的磁盘里必须是干净的，里面没有数据。
- 下面示例里的IP仅供示例参考，你需要改成你真实用到的IP和文件夹路径。
- 分布式Minio里的节点时间差不能超过3秒，你可以使用[NTP](#) 来保证时间一致。
- 在Windows下运行分布式Minio处于实验阶段，请悠着点使用。

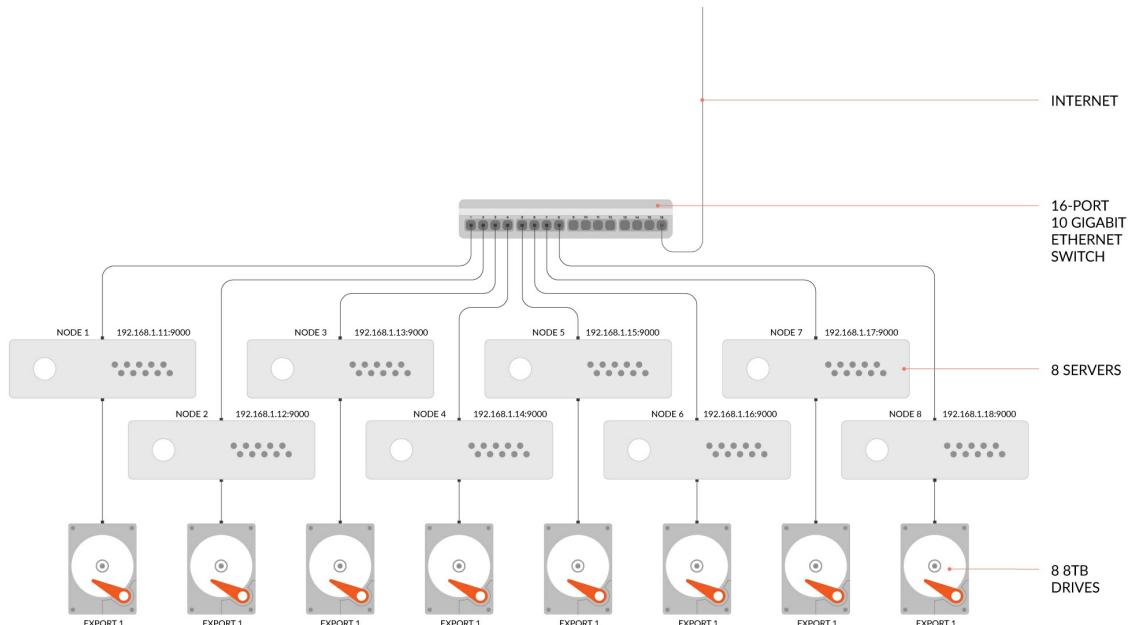
示例1: 启动分布式Minio实例, 8个节点, 每节点1块盘, 需要在8个节点上都运行下面的命令。

GNU/Linux 和 macOS

```
export MINIO_ACCESS_KEY=<ACCESS_KEY>
export MINIO_SECRET_KEY=<SECRET_KEY>
minio server http://192.168.1.11/export1 http://192.168.1.12/export2 \
    http://192.168.1.13/export3 http://192.168.1.14/export4 \
    http://192.168.1.15/export5 http://192.168.1.16/export6 \
    http://192.168.1.17/export7 http://192.168.1.18/export8
```

Windows

```
set MINIO_ACCESS_KEY=<ACCESS_KEY>
set MINIO_SECRET_KEY=<SECRET_KEY>
minio.exe server http://192.168.1.11/C:/data http://192.168.1.12/C:/data ^
    http://192.168.1.13/C:/data http://192.168.1.14/C:/data ^
    http://192.168.1.15/C:/data http://192.168.1.16/C:/data ^
    http://192.168.1.17/C:/data http://192.168.1.18/C:/data
```



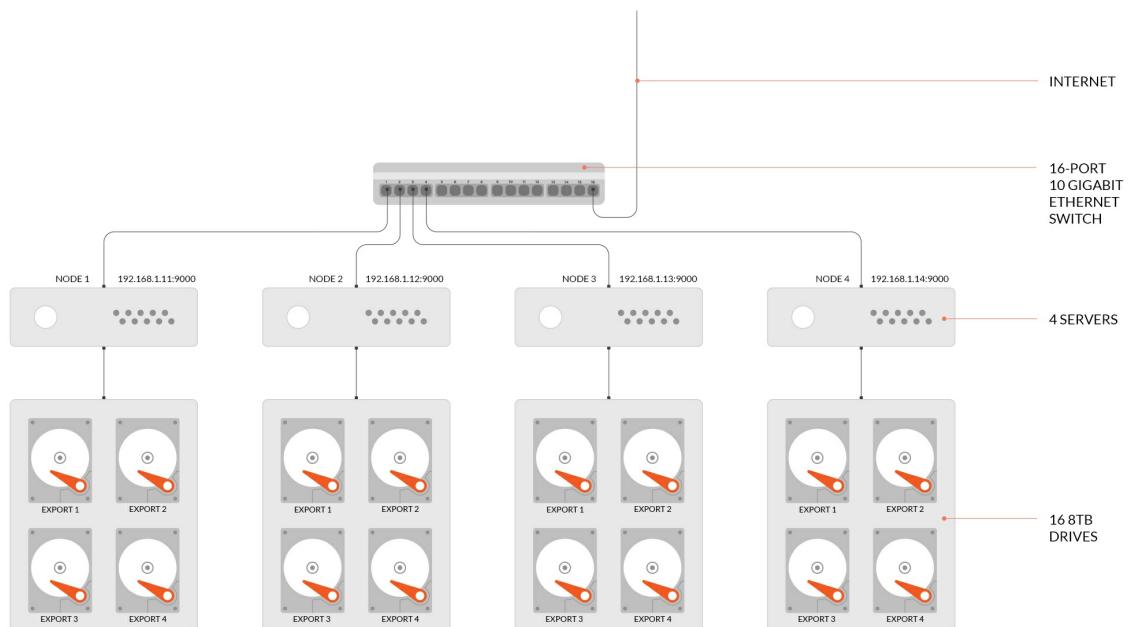
示例2: 启动分布式Minio实例, 4个节点, 每节点4块盘, 需要在4个节点上都运行下面的命令。

GNU/Linux 和 macOS

```
export MINIO_ACCESS_KEY=<ACCESS_KEY>
export MINIO_SECRET_KEY=<SECRET_KEY>
minio server http://192.168.1.11/export1 http://192.168.1.11/export2 \
    http://192.168.1.11/export3 http://192.168.1.11/export4 \
    http://192.168.1.12/export1 http://192.168.1.12/export2 \
    http://192.168.1.12/export3 http://192.168.1.12/export4 \
    http://192.168.1.13/export1 http://192.168.1.13/export2 \
    http://192.168.1.13/export3 http://192.168.1.13/export4 \
    http://192.168.1.14/export1 http://192.168.1.14/export2 \
    http://192.168.1.14/export3 http://192.168.1.14/export4
```

Windows

```
set MINIO_ACCESS_KEY=<ACCESS_KEY>
set MINIO_SECRET_KEY=<SECRET_KEY>
minio.exe server http://192.168.1.11/C:/data1 http://192.168.1.11/C:/data2 ^
    http://192.168.1.11/C:/data3 http://192.168.1.11/C:/data4 ^
    http://192.168.1.12/C:/data1 http://192.168.1.12/C:/data2 ^
    http://192.168.1.12/C:/data3 http://192.168.1.12/C:/data4 ^
    http://192.168.1.13/C:/data1 http://192.168.1.13/C:/data2 ^
    http://192.168.1.13/C:/data3 http://192.168.1.13/C:/data4 ^
    http://192.168.1.14/C:/data1 http://192.168.1.14/C:/data2 ^
    http://192.168.1.14/C:/data3 http://192.168.1.14/C:/data4
```



扩展现有的分布式集群

例如我们是通过区的方式启动MinIO集群，命令行如下：

```
export MINIO_ACCESS_KEY=<ACCESS_KEY>
export MINIO_SECRET_KEY=<SECRET_KEY>
minio server http://host{1...32}/export{1...32}
```

MinIO支持通过命令，指定新的集群来扩展现有集群（纠删码模式），命令行如下：

```
export MINIO_ACCESS_KEY=<ACCESS_KEY>
export MINIO_SECRET_KEY=<SECRET_KEY>
minio server http://host{1...32}/export{1...32} http://host{33...64}/export{1...32}
```

现在整个集群就扩展了1024个磁盘，总磁盘变为2048个，新的对象上传请求会自动分配到最少使用的集群上。通过以上扩展策略，您就可以按需扩展您的集群。重新配置后重启集群，会立即在集群中生效，并对现有集群无影响。如上命令中，我们可以把原来的集群看做一个区，新增集群看做另一个区，新对象按每个区域中的可用空间比例放置在区域中。在每个区域内，基于确定性哈希算法确定位置。

说明：您添加的每个区域必须具有与原始区域相同的磁盘数量（纠删码集）大小，以便维持相同的数据冗余**SLA**。例如，第一个区有8个磁盘，您可以将集群扩展为16个、32个或1024个磁盘的区域，您只需确保部署的**SLA**是原始区域的倍数即可。

3. 验证

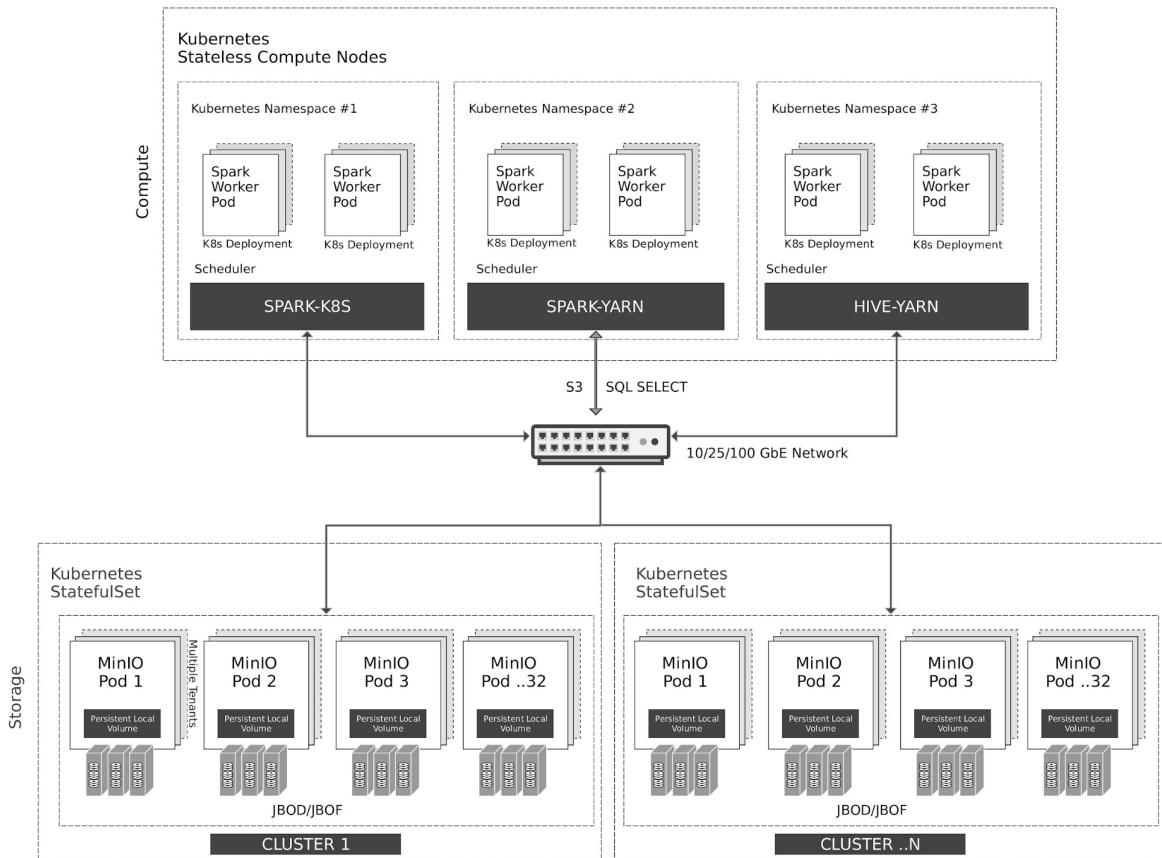
验证是否部署成功，使用浏览器访问Minio服务或者使用 `mc`。多个节点的存储容量和就是分布式Minio的存储容量。

了解更多

- [Minio纠删码快速入门](#)
- 使用 `mc`
- 使用 `aws-cli`
- 使用 `s3cmd`
- 使用 `minio-go` SDK
- [minio官方文档](#)

使用MinIO分解的HDP Spark和Hive

1. 云原生架构



Kubernetes在计算节点上弹性地管理无状态的Spark和Hive容器。Spark具有与Kubernetes的本机调度程序集成。由于传统原因，Hive在Kubernetes上使用YARN调度程序。

通过S3 / SQL SELECT API对MinIO对象存储的所有访问。除了计算节点外，Kubernetes还将MinIO容器作为有状态容器进行管理，其状态存储（JBOD / JBOF）映射为持久性本地卷。这种架构支持多租户MinIO，从而可以隔离客户之间的数据。

MinIO还支持类似于AWS区域和层的多集群，多站点联合。使用MinIO信息生命周期管理（ILM），可以将数据配置为在基于NVMe的热存储和基于HDD的热存储之间分层。所有数据均使用每个对象的密钥加密。MinIO使用OpenID Connect或Kerberos / LDAP / AD对租户之间的访问控制和身份管理进行管理。

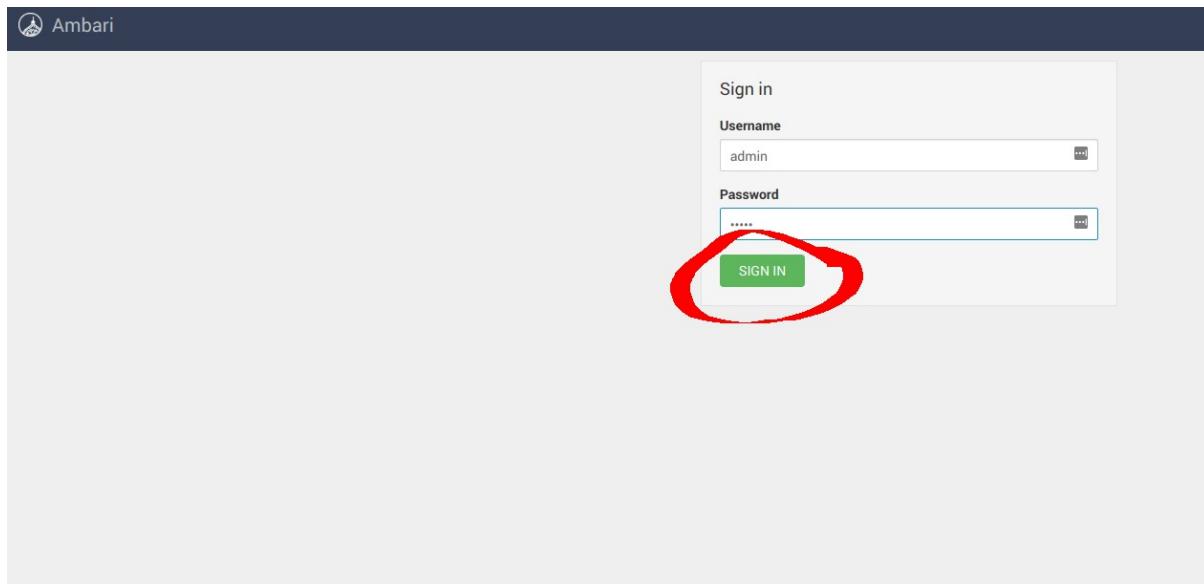
2. 先决条件

- 使用本 guide.
- 安装Hortonworks Distribution
- 设置 Ambari 它会自动设置YARN
 - 安装 Spark

- 使用以下指南之一安装MinIO Distributed Server。
 - 基于Kubernetes的部署
 - 基于MinIO Helm Chart的部署

3. 配置Hadoop, Spark, Hive以使用MinIO

A成功安装后，导航到Ambari UI <http://:8080>并使用默认凭据登录：[用户名：**admin**, 密码：**admin**]



3.1 配置Hadoop

导航到服务 -> HDFS -> CONFIGS -> 高级 如下所示

 A screenshot of the Ambari Services / HDFS / Configs page. The left sidebar shows various services like HDFS, YARN, and MapReduce2, with 'HDFS' highlighted by a red circle. The main area shows the 'CONFIGS' tab selected. Under 'SETTINGS' (ADVANCED), there are two sections: 'NameNode' and 'Secondary NameNode'. In 'NameNode', settings include 'NameNode host' (hdp-cl1), 'NameNode new generation size' (384 MB), 'NameNode maximum new generation size' (384 MB), 'NameNode permanent generation size' (384 MB), and 'NameNode maximum permanent generation size' (384 MB). In 'Secondary NameNode', settings include 'SNameNode host' (hdp-cl2) and 'SecondaryNameNode Checkpoint directories' (/hadoop/hdfs/namesecondary).

NameNode	
NameNode host	hdp-cl1
NameNode new generation size	384 MB
NameNode maximum new generation size	384 MB
NameNode permanent generation size	384 MB
NameNode maximum permanent generation size	384 MB

Secondary NameNode	
SNameNode host	hdp-cl2
SecondaryNameNode Checkpoint directories	/hadoop/hdfs/namesecondary

导航到 自定义核心网站 为 `_s3a_` 连接器配置MinIO参数

The screenshot shows the Ambari web interface with the HDFS service selected in the sidebar. On the right, the 'Advanced' section of the HDFS configuration is displayed. A red circle highlights the 'Custom core-site' option, which is listed among other advanced configuration options like 'Advanced viewfs-mount-table' and 'Custom hadoop-metrics2.properties'.

```
sudo pip install yq
alias kv-pairify='xq ".configuration[]" | jq ".[]" | jq -r ".name + \"=\\" + .value\""
```

让我们以12个计算节点的集合（聚合内存为1.2TiB）为例，我们需要进行以下设置以获得最佳结果。以下最佳项添加核心的site.xml配置S3A与MinIO。这里最重要的选择是

```
cat ${HADOOP_CONF_DIR}/core-site.xml | kv-pairify | grep "mapred"

mapred.maxthreads.generate.mapoutput=2 # Num threads to write map outputs
mapred.maxthreads.partition.closer=0 # Asynchronous map flushers
mapreduce.fileoutputcommitter.algorithm.version=2 # Use the latest committer version
mapreduce.job.reduce.slowstart.completedmaps=0.99 # 99% map, then reduce
mapreduce.reduce.shuffle.input.buffer.percent=0.9 # Min % buffer in RAM
mapreduce.reduce.shuffle.merge.percent=0.9 # Minimum % merges in RAM
mapreduce.reduce.speculative=false # Disable speculation for reducing
mapreduce.task.io.sort.factor=999 # Threshold before writing to disk
mapreduce.task.sort.spill.percent=0.9 # Minimum % before spilling to disk
```

S3A是使用S3和其他与S3兼容的对象存储库（如MinIO）的连接器。MapReduce工作负载通常以与HDFS相同的方式与对象存储交互。这些工作负载依靠HDFS原子重命名功能来完成将数据写入数据存储中。对象存储操作本质上是原子的，不需要/实现重命名API。默认的S3A提交者通过复制和删除API模拟重命名。由于写入放大，这种交互模式会导致性能显著下降。奈飞例如，开发了两个新的登台提交程序-目录登台提交程序和分区登台提交程序-以充分利用本机对象存储操作。这些提交者不需要重命名操作。评估了两个登台提交者，以及另一个称为基准的Magic提交者。

发现目录登台提交程序是三个中最快的，S3A连接器应配置以下参数以获得最佳结果：

```
cat ${HADOOP_CONF_DIR}/core-site.xml | kv-pairify | grep "s3a"

fs.s3a.access.key=minio
fs.s3a.secret.key=minio123
fs.s3a.path.style.access=true
```

```

fs.s3a.block.size=512M
fs.s3a.buffer.dir=${hadoop.tmp.dir}/s3a
fs.s3a.committer.magic.enabled=false
fs.s3a.committer.name=directory
fs.s3a.committer.staging.abort.pending.uploads=true
fs.s3a.committer.staging.conflict-mode=append
fs.s3a.committer.staging.tmp.path=/tmp/staging
fs.s3a.committer.staging.unique-filenames=true
fs.s3a.connection.establish.timeout=5000
fs.s3a.connection.ssl.enabled=false
fs.s3a.connection.timeout=20000
fs.s3a.endpoint=http://minio:9000
fs.s3a.impl=org.apache.hadoop.fs.s3a.S3AFileSystem

fs.s3a.committer.threads=2048 # Number of threads writing to MinIO
fs.s3a.connection.maximum=8192 # Maximum number of concurrent connections
fs.s3a.fast.upload.active.blocks=2048 # Number of parallel uploads
fs.s3a.fast.upload.buffer=disk # Use disk as the buffer for uploads
fs.s3a.fast.upload=true # Turn on fast upload mode
fs.s3a.max.total.tasks=2048 # Maximum number of parallel tasks
fs.s3a.multipart.size=512M # Size of each multipart chunk
fs.s3a.multipart.threshold=512M # Size before using multipart uploads
fs.s3a.socket.recv.buffer=65536 # Read socket buffer hint
fs.s3a.socket.send.buffer=65536 # Write socket buffer hint
fs.s3a.threads.max=2048 # Maximum number of threads for S3A

```

其他优化选项的其余部分将在下面的链接中讨论

- <https://hadoop.apache.org/docs/current/hadoop-aws/tools/hadoop-aws/index.html>
- <https://hadoop.apache.org/docs/r3.1.1/hadoop-aws/tools/hadoop-aws/committers.html>

应用配置更改后，继续重新启动 **Hadoop** 服务。

The screenshot shows the Cloudera Manager web interface for managing HDFS services. The top navigation bar includes links for tpch, settings, metrics, and a user dropdown. Below the navigation is a breadcrumb trail: Home / Services / HDFS / Configs. The main content area has tabs for SUMMARY, HEATMAPS, CONFIGS (which is active), and METRICS. On the left, there's a sidebar with SETTINGS and ADVANCED tabs. The main panel is divided into two sections: NameNode and DataNode. Each section contains several configuration parameters with edit icons. To the right of the configuration panels is a large 'ACTIONS' dropdown menu. One item in this menu, 'Restart All', is highlighted with a red circle.

3.2 配置Spark2

导航到服务 -> **Spark2** -> **CONFIGS**，如下所示

The screenshot shows the Ambari interface. On the left, the sidebar has 'Services' expanded, with 'Spark2' highlighted and circled in red. The main content area is titled 'Services / Spark2 / Configs'. It shows the 'CONFIGS' tab is active. A dropdown menu shows 'Version: 2'. Below it is a list of configuration files:

- Advanced livy2-client-conf
- Advanced livy2-conf
- Advanced livy2-env
- Advanced livy2-log4j-properties
- Advanced livy2-spark-blacklist
- Advanced spark2-defaults
- Advanced spark2-env
- Advanced spark2-hive-site-override

导航到“Custom spark-defaults”以配置s3a连接器的MinIO参数

The screenshot shows the Ambari interface. On the left, the sidebar has 'Services' expanded, with 'Spark2' highlighted and circled in green. The main content area is titled 'Services / Spark2 / Configs'. It shows the 'CONFIGS' tab is active. A dropdown menu shows 'Version: 2'. Below it is a list of configuration files, with 'Custom spark2-defaults' and its 'Add Property ...' link circled in red.

- Advanced livy2-spark-blacklist
- Advanced spark2-defaults
- Advanced spark2-env
- Advanced spark2-hive-site-override
- Advanced spark2-log4j-properties
- Advanced spark2-metrics-properties
- Advanced spark2-thrift-fairscheduler
- Advanced spark2-thrift-sparkconf
- Custom livy2-client-conf
- Custom livy2-conf
- Custom spark2-defaults**
- Add Property ...
- Custom spark2-hive-site-override
- Custom spark2-metrics-properties
- Custom spark2-thrift-fairscheduler
- Custom spark2-thrift-sparkconf

为spark-defaults.conf添加以下最佳条目，以使用 MinIO 配置Spark。

```
spark.hadoop.fs.s3a.access.key minio
spark.hadoop.fs.s3a.secret.key minio123
spark.hadoop.fs.s3a.path.style.access true
spark.hadoop.fs.s3a.block.size 512M
spark.hadoop.fs.s3a.buffer.dir ${hadoop.tmp.dir}/s3a
```

```

spark.hadoop.fs.s3a.committer.magic.enabled false
spark.hadoop.fs.s3a.committer.name directory
spark.hadoop.fs.s3a.committer.staging.abort.pending.uploads true
spark.hadoop.fs.s3a.committer.staging.conflict-mode append
spark.hadoop.fs.s3a.committer.staging.tmp.path /tmp/staging
spark.hadoop.fs.s3a.committer.staging.unique-filenames true
spark.hadoop.fs.s3a.committer.threads 2048 # number of threads writing to MinIO
spark.hadoop.fs.s3a.connection.establish.timeout 5000
spark.hadoop.fs.s3a.connection.maximum 8192 # maximum number of concurrent conns
spark.hadoop.fs.s3a.connection.ssl.enabled false
spark.hadoop.fs.s3a.connection.timeout 200000
spark.hadoop.fs.s3a.endpoint http://minio:9000
spark.hadoop.fs.s3a.fast.upload.active.blocks 2048 # number of parallel uploads
spark.hadoop.fs.s3a.fast.upload.buffer disk # use disk as the buffer for uploads
spark.hadoop.fs.s3a.fast.upload true # turn on fast upload mode
spark.hadoop.fs.s3a.impl org.apache.hadoop.spark.hadoop.fs.s3a.S3AFileSystem
spark.hadoop.fs.s3a.max.total.tasks 2048 # maximum number of parallel tasks
spark.hadoop.fs.s3a.multipart.size 512M # size of each multipart chunk
spark.hadoop.fs.s3a.multipart.threshold 512M # size before using multipart uploads
spark.hadoop.fs.s3a.socket.recv.buffer 65536 # read socket buffer hint
spark.hadoop.fs.s3a.socket.send.buffer 65536 # write socket buffer hint
spark.hadoop.fs.s3a.threads.max 2048 # maximum number of threads for S3A

```

应用配置更改后，继续重新启动 **Spark** 服务。

The screenshot shows the Cloudera Manager interface for managing services. In the top navigation bar, there are links for tpch, configuration, logs, and metrics, along with a user dropdown for 'admin'. Below the navigation is a search bar and a dashboard view. The main area is titled 'Services / Spark2 / Configs'. It has tabs for 'SUMMARY' and 'CONFIGS', with 'CONFIGS' currently selected. A list of configuration files is shown, including 'Advanced livy2-client-conf', 'Advanced livy2-conf', 'Advanced livy2-env', 'Advanced livy2-log4j-properties', 'Advanced livy2-spark-blacklist', and 'Advanced spark2-defaults'. To the right of the configuration list is an 'ACTIONS' dropdown menu. This menu contains several options: 'Start', 'Stop', 'Restart All' (which is circled in red), 'Run Service Check', 'Turn On Maintenance Mode', 'Download Client Configs', and 'Delete Service'.

3.3 配置Hive

导航到 服务 -> **Hive** -> **CONFIGS-> ADVANCED**， 如下所示

Ambari / Services / Hive / Configs

SUMMARY CONFIGS

Version: 4

SETTINGS DATABASE ADVANCED

Hive Metastore

Hive Metastore host: hdp-cl3

General

datanucleus.cache.level2.type	none
hive.compactor.check.interval	300
hive.compactor.delta.num.threshold	10
hive.compactor.delta.pct.threshold	0.1f
Run Compactor	<input checked="" type="checkbox"/> <input type="button"/> <input type="button"/> <input type="button"/>

导航到“自定义配置单元站点”以配置s3a连接器的MinIO参数

Ambari / Services / ... / Hive / Configs

Custom hive-exec-log4j2

Custom hive-interactive-site

Custom hive-log4j2

Custom hive-site

Add Property ...

Custom hivemetastore-site

Custom hiveserver2-interactive-site

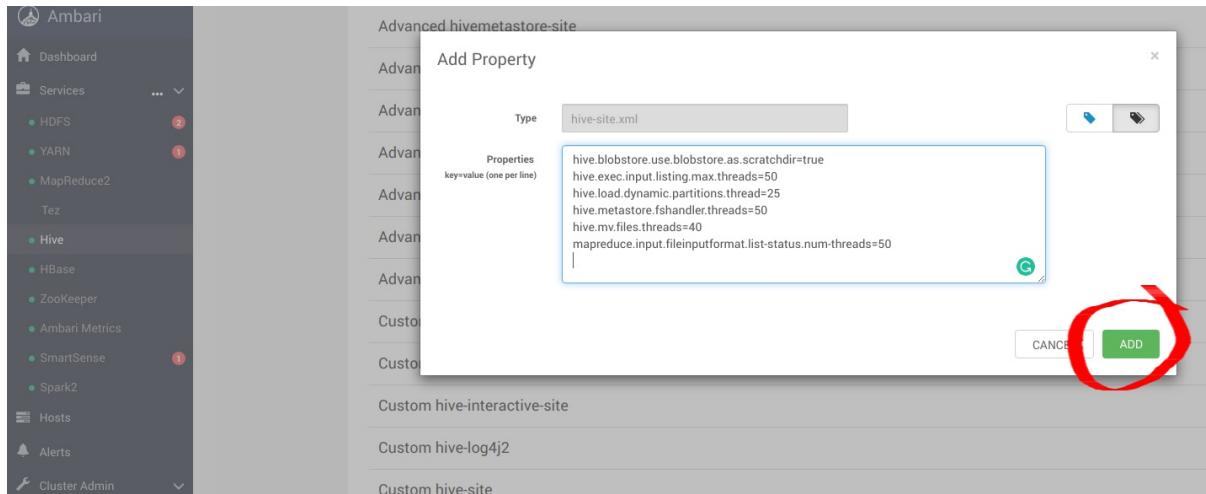
Custom hiveserver2-site

Custom llap-cli-log4j2

添加以下最佳条目hive-site.xml以使用 MinIO 配置Hive。.

```
hive.blobstore.use.blobstore.as.scratchdir=true
hive.exec.input.listing.max.threads=50
hive.load.dynamic.partitions.thread=25
hive.metastore.fshandler.threads=50
hive.mv.files.threads=40
mapreduce.input.fileinputformat.list-status.num-threads=50
```

有关这些选项的更多信息，请访问 https://www.cloudera.com/documentation/enterprise/5-11-x/topics/admin_hive_on_s3_tuning.html



应用配置更改后，继续重新启动所有Hive服务。

4. 运行示例应用程序

成功安装Hive, Hadoop和Spark之后，我们现在可以继续运行一些示例应用程序，以查看它们是否配置正确。我们可以使用Spark Pi和Spark WordCount程序来验证我们的Spark安装。我们还可以探索如何从命令行和Spark Shell运行Spark作业。

4.1 Spark Pi

通过运行以下计算密集型示例来测试Spark安装，该示例通过将“飞镖”围成一圈来计算pi。该程序在单位平方 ((0, 0) 到 (1, 1)) 中生成点，并计算在平方内单位圆内有多少点。结果近似为pi。

请按照以下步骤运行Spark Pi示例：

- 以用户 ‘spark’ 登录。
- 当作业运行时，库现在可以在中间处理期间使用MinIO。
- 使用Spark客户端导航到节点，并访问spark2-client目录：

```
cd /usr/hdp/current/spark2-client
su spark
```

- 使用 **org.apache.spark** 中的代码在yarn-client模式下运行Apache Spark Pi作业：

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
--master yarn-client \
--num-executors 1 \
--driver-memory 512m \
```

```
--executor-memory 512m \
--executor-cores 1 \
examples/jars/spark-examples*.jar 10
```

该作业应产生如下所示的输出。注意输出中的pi值。

```
17/03/22 23:21:10 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.scala:38, took 1.302805 s  
Pi is roughly 3.1445191445191445
```

通过浏览器到YARN ResourceManager Web UI并单击作业历史记录服务器信息，也可以在浏览器中查看作业状态。

4.2 字数统计

WordCount是一个简单的程序，可计算单词在文本文件中出现的频率。该代码构建一个称为counts的（字符串，整数）对的数据集，并将该数据集保存到文件中。

下面的示例将WordCount代码提交到Scala shell。为Spark WordCount示例选择一个输入文件。我们可以使用任何文本文件作为输入。

- 以用户 ‘**spark**’ 登录
 - 当作业运行时，库现在可以在中间处理期间使用**MinIO**。
 - 使用**Spark**客户端导航到节点，并访问**spark2-client**目录：

```
cd /usr/hdp/current/spark2-client  
su spark
```

s3a://testbucket/testdata

4.2.2 运行Spark shell:

```
./bin/spark-shell --master yarn-client --driver-memory 512m --executor-memory 512m
```

该命令将产生如下所示的输出。（带有其他状态消息）：

```
Spark context Web UI available at http://172.26.236.247:4041
Spark context available as 'sc' (master = yarn, app id = application_1490217230866_0002).
Spark session available as 'spark'.
Welcome to
```

```
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_112)
Type in expressions to have them evaluated.
Type :help for more information.
```

scala>

- 在**scala>**提示符下，通过键入以下命令（用值替换节点名，文件名和文件位置）来提交作业：

```
scala> val file = sc.textFile("s3a://testbucket/testdata")
file: org.apache.spark.rdd.RDD[String] = s3a://testbucket/testdata MapPartitionsRDD[1] at textFile at <console>:24

scala> val counts = file.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:25

scala> counts.saveAsTextFile("s3a://testbucket/wordcount")
```

使用以下方法之一查看作业输出：

在**Scala shell**中查看输出：

```
scala> counts.count()
364
```

要查看MinIO的输出，请退出**Scala shell**。查看WordCount作业状态：

```
hadoop fs -ls s3a://testbucket/wordcount
```

输出应类似于以下内容：

```
Found 3 items
-rw-rw-rw- 1 spark spark      0 2019-05-04 01:36 s3a://testbucket/wordcount/_SUCCESS
-rw-rw-rw- 1 spark spark    4956 2019-05-04 01:36 s3a://testbucket/wordcount/part-00000
-rw-rw-rw- 1 spark spark    5616 2019-05-04 01:36 s3a://testbucket/wordcount/part-00001
```

使用TLS安全的访问Minio服务

本文，我们讲介绍如何在Linux和Windows上配置Minio服务使用TLS。

1. 前提条件

- 下载Minio server [这里](#)

2. 配置已存在的证书

如果你已经有私钥和公钥证书，你需要将它们拷贝到Minio的config/ certs 文件夹,分别取名为 private.key 和 public.crt 。

如果这个证书是被证书机构签发的， public.crt 应该是服务器的证书，任何中间体的证书以及CA的根证书的级联。

3. 生成证书

Linux

Minio在Linux只支持使用PEM格式的key/certificate。

使用 Let's Encrypt

更多信息，请访问 [这里](#)

使用 generate_cert.go (self-signed certificate)

你需要下载 [generate_cert.go](#)，它是一个简单的go工具，可以生成自签名的证书，不过大多数情况下用着都是木有问题的。

generate_cert.go 已经提供了带有DNS和IP条目的SAN证书:

```
go run generate_cert.go -ca --host "10.10.0.3"
```

使用 OpenSSL:

生成私钥:

```
openssl genrsa -out private.key 2048
```

生成自签名证书:

```
openssl req -new -x509 -days 3650 -key private.key -out public.crt -subj "/C=US/ST=state/L=location/O=organization/CN=domain"
```

Windows

Minio在Windows上只支持PEM格式的key/certificate，目前不支持PFX证书。

安装 GnuTLS

下载并解压[GnuTLS](#)

确保将解压后的binary路径加入到系统路径中。

```
setx path "%path%;C:\Users\MyUser\Downloads\gnutls-3.4.9-w64\bin"
```

你可能需要重启powershell控制台来使其生效。

生成**private.key**

运行下面的命令来生成 `private.key`

```
certtool.exe --generate-privkey --outfile private.key
```

生成**public.crt**

创建文件 `cert.cnf`， 填写必要信息来生成证书。

```
# X.509 Certificate options
#
# DN options

# The organization of the subject.
organization = "Example Inc."

# The organizational unit of the subject.
#unit = "sleeping dept."

# The state of the certificate owner.
state = "Example"

# The country of the subject. Two letter code.
country = "EX"

# The common name of the certificate owner.
cn = "Sally Certowner"

# In how many days, counting from today, this certificate will expire.
expiration_days = 365

# X.509 v3 extensions

# DNS name(s) of the server
dns_name = "localhost"

# (Optional) Server IP address
ip_address = "127.0.0.1"

# Whether this certificate will be used for a TLS server
tls_www_server

# Whether this certificate will be used to encrypt data (needed
# in TLS RSA ciphersuites). Note that it is preferred to use different
# keys for encryption and signing.
encryption_key
```

生成公钥证书

```
certtool.exe --generate-self-signed --load-privkey private.key --template cert.cnf --outfile public.crt
```

4. 安装第三方CAs

Minio可以配置成连接其它服务，不管是Minio节点还是像NATs、Redis这些。如果这些服务用的不是在已知证书机构注册的证书，你可以让Minio服务信任这些CA，怎么做呢，将这些证书放到Minio配置路径下(`~/.minio/certs/CAs/` Linux 或者 `C:\Users\<Username>\.minio\certs\CAs` Windows).

了解更多

- [Minio快速入门](#)
- [Minio客户端权威指南](#)

MinIO存储桶通知指南

存储桶（Bucket）如果发生改变,比如上传对象和删除对象, 可以使用存储桶事件通知机制进行监控, 并通过以下方式发布出去:

Notification Targets
AMQP
MQTT
Elasticsearch
Redis
NATS
PostgreSQL
MySQL
Apache Kafka
Webhooks

前提条件

- 从[这里](#)下载并安装MinIO Server。
- 从[这里](#)下载并安装MinIO Client。

使用AMQP发布MinIO事件

从[这里](#)下载安装RabbitMQ。

第一步: 将AMQP endpoint添加到MinIO

MinIO Server的配置文件默认路径是 `~/.minio/config.json`。AMQP配置信息是在 `notify` 这个节点下的 `amqp` 节点下, 在这里为你的AMQP实例创建配置信息键值对, `key`是你的AMQP endpoint的名称, `value`是下面表格中列列的键值对集合。

参数	类型	描述
enable	bool	(必须) 此AMQP server endpoint是否可用
url	string	(必须) AMQP server endpoint, 例如. <code>amqp://myuser:mypassword@localhost:5672</code>
exchange	string	exchange名称
routingKey	string	发布用的Routing key
exchangeType	string	exchange类型
deliveryMode	uint8	发布方式。0或1 - 瞬态; 2 - 持久。
mandatory	bool	Publishing related bool.
immediate	bool	Publishing related bool.
durable	bool	Exchange declaration related bool.
internal	bool	Exchange declaration related bool.
nowait	bool	Exchange declaration related bool.
autoDeleted	bool	Exchange declaration related bool.

下面展示的是RabbitMQ的配置示例:

```

"amqp": {
  "1": {
    "enable": true,
    "url": "amqp://myuser:mypassword@localhost:5672",
    "exchange": "bucketevents",
    "routingKey": "bucketlogs",
    "exchangeType": "fanout",
    "deliveryMode": 0,
    "mandatory": false,
    "immediate": false,
    "durable": false,
    "internal": false,
    "noWait": false,
    "autoDeleted": false
  }
}

```

更新完配置文件后，重启MinIO Server让配置生效。如果一切顺利，MinIO Server会在启动时输出一行信息，类似 `SQS ARNs: arn:minio:sqs:us-east-1:1:amqp`。

MinIO支持[RabbitMQ](#)中所有的交换方式，这次我们采用 `fanout` 交换。

注意一下，你可以听从你内心的想法，想配几个AMQP服务就配几个，只要每个AMQP服务实例有不同的ID (比如前面示例中的“1”) 和配置信息。

第二步：使用MinIO客户端启用bucket通知

如果一个JPEG图片上传到 `myminio server`里的 `images` 存储桶或者从桶中删除，一个存储桶事件通知就会被触发。这里 ARN值是 `arn:minio:sqs:us-east-1:1:amqp`，想了解更多关于ARN的信息，请参考[AWS ARN documentation](#).

```

mc mb myminio/images
mc event add myminio/images arn:minio:sqs:us-east-1:1:amqp --suffix .jpg
mc event list myminio/images
arn:minio:sqs:us-east-1:1:amqp s3:ObjectCreated:* s3:ObjectRemoved:* Filter: suffix=".jpg"

```

第三步：在RabbitMQ上进行验证

下面将要出场的python程序会等待队列交换T `bucketevents` 并在控制台中输出事件通知。我们使用的是[Pika Python Client](#)来实现此功能。

```

#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(
    host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='bucketevents',
                        exchange_type='fanout')

result = channel.queue_declare(exclusive=False)
queue_name = result.method.queue

channel.queue_bind(exchange='bucketevents',
                  queue=queue_name)

print(' [*] Waiting for logs. To exit press CTRL+C')

```

```

def callback(ch, method, properties, body):
    print(" [x] %r" % body)

channel.basic_consume(callback,
                      queue=queue_name,
                      no_ack=False)

channel.start_consuming()

```

执行示例中的python程序来观察RabbitMQ事件。

```
python rabbit.py
```

另开一个terminal终端并上传一张JPEG图片到 `images` 存储桶。

```
mc cp myphoto.jpg myminio/images
```

一旦上传完毕，你应该会通过RabbitMQ收到下面的事件通知。

```

python rabbit.py
[{"Records": [{"eventVersion": "2.0", "eventSource": "aws:s3", "awsRegion": "us-east-1", "eventTime": "2016-09-08T22:34:38.226Z", "eventName": "s3:ObjectCreated:Put", "userIdentity": {"principalId": "minio"}, "requestParameters": {"sourceIPAddress": "10.1.10.150:44576"}, "responseElements": {}, "s3": {"s3SchemaVersion": "1.0", "configurationId": "Config", "bucket": {"name": "images"}, "ownerIdentity": {"principalId": "minio"}, "arn": "arn:aws:s3:::images", "object": {"key": "myphoto.jpg", "size": 200436, "sequencer": "147279EA9F40933"}}, "level": "info", "msg": "", "time": "2016-09-08T15:34:38-07:00"}]\n

```

使用MQTT发布MinIO事件

从[这里](#)安装MQTT Broker。

第一步：添加MQTT endpoint到MinIO

MinIO Server的配置文件默认路径是 `~/.minio/config.json`。MQTT配置信息是在 `notify` 这个节点下的 `mqtt` 节点下，在这里为你的MQTT实例创建配置信息键值对，`key`是你的MQTT endpoint的名称，`value`是下面表格中列列的键值对集合。

参数	类型	描述
<code>enable</code>	<code>bool</code>	(必须)这个server endpoint是否可用?
<code>broker</code>	<code>string</code>	(必须)MQTT server endpoint, 例如. <code>tcp://localhost:1883</code>
<code>topic</code>	<code>string</code>	(必须)要发布的MQTT主题的名称, 例如. <code>minio</code>
<code>qos</code>	<code>int</code>	设置服务质量级别
<code>clientId</code>	<code>string</code>	MQTT代理识别MinIO的唯一ID
<code>username</code>	<code>string</code>	连接MQTT server的用户名(如果需要的话)
<code>password</code>	<code>string</code>	链接MQTT server的密码(如果需要的话)

以下是一个MQTT的配置示例:

```

"mqtt": {
    "1": {
        "enable": true,
        "broker": "tcp://localhost:1883",

```

```

        "topic": "minio",
        "qos": 1,
        "clientId": "minio",
        "username": "",
        "password": ""
    }
}

```

更新完配置文件后，重启MinIO Server让配置生效。如果一切顺利，MinIO Server会在启动时输出一行信息，类似 `SQS ARNs: arn:minio:sqs:us-east-1:1:mqtt`。

MinIO支持任何支持MQTT 3.1或3.1.1的MQTT服务器，并且可以通过TCP, TLS或WebSocket连接使用 `tcp://`, `tls://`, or `ws://` 分别作为代理URL的方案。更多信息，请参考 [Go Client](#)。

注意一下，你还是和之前AMQP一样可以听从你内心的想法，想配几个MQTT服务就配几个，只要每个MQTT服务实例有不同的ID(比如前面示例中的“1”)和配置信息。

第二步：使用MinIO客户端启用bucket通知

如果一个JPEG图片上传到 `myminio server`里的 `images` 存储桶或者从桶中删除，一个存储桶事件通知就会被触发。这里ARN值是 `arn:minio:sqs:us-east-1:1:mqtt`。

```

mc mb myminio/images
mc event add myminio/images arn:minio:sqs:us-east-1:1:mqtt --suffix .jpg
mc event list myminio/images
arn:minio:sqs:us-east-1:1:amqp s3:ObjectCreated:* s3:ObjectRemoved:* Filter: suffix=".jpg"

```

第三步：验证MQTT

下面的python程序等待mqtt主题 / minio，并在控制台上打印事件通知。我们使用[paho-mqtt](#)库来执行此操作。

```

#!/usr/bin/env python
from __future__ import print_function
import paho.mqtt.client as mqtt

# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):
    print("Connected with result code", rc)

    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe("/minio")

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    print(msg.payload)

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("localhost:1883", 1883, 60)

# Blocking call that processes network traffic, dispatches callbacks and
# handles reconnecting.
# Other loop*() functions are available that give a threaded interface and a
# manual interface.
client.loop_forever()

```

执行这个python示例程序来观察MQTT事件。

```
python mqtt.py
```

打开一个新的terminal终端并上传一张JPEG图片到 `images` 存储桶。

```
mc cp myphoto.jpg myminio/images
```

一旦上传完毕，你应该会通过MQTT收到下面的事件通知。

```
python mqtt.py
[{"Records": [{"eventVersion": "2.0", "eventSource": "aws:s3", "awsRegion": "us-east-1", "eventTime": "2016-09-08T22:34:38.226Z", "eventName": "s3:ObjectCreated:Put", "userIdentity": {"principalId": "minio"}, "requestParameters": {"sourceIPAddress": "10.1.10.150:44576"}, "responseElements": {}, "s3": {"s3SchemaVersion": "1.0", "configurationId": "Config", "bucket": {"name": "images"}, "ownerIdentity": {"principalId": "minio"}, "arn": "arn:aws:s3:::images", "object": {"key": "myphoto.jpg", "size": 200436, "sequencer": "147279EAF9F40933"}}], "level": "info", "msg": "", "time": "2016-09-08T15:34:38-07:00"}
```

使用Elasticsearch发布MinIO事件

安装 [Elasticsearch](#)。

这个通知目标支持两种格式: `namespace` and `access`。

如果使用的是 `namespace` 格式, MinIO将桶中的对象与索引中的文档进行同步。对于MinIO的每一个事件, ES都会创建一个`document`,这个`document`的ID就是存储桶以及存储对象的名称。事件的其他细节存储在`document`的正文中。因此, 如果一个已经存在的对象在MinIO中被覆盖, 在ES中的相对应的`document`也会被更新。如果一个对象被删除, 相对应的`document`也会从index中删除。

如果使用的是`access`格式, MinIO将事件作为`document`加到ES的`index`中。对于每一个事件, ES同样会创建一个`document`,这个`document`包含事件的所有细节, `document`的时间戳设置为事件的时间戳, 并将该`document`加到ES的`index`中。这个`document`的ID是由ES随机生成的。在`access`格式下, 没有文档会被删除或者修改, 对于一个对象的操作, 都会生成新的`document`附加到`index`中。

下面的步骤展示的是在 `namespace` 格式下, 如何使用通知目标。另一种格式和这个很类似, 为了不让你们说我墨迹, 就不再赘述了。

第一步：确保至少满足第低要求

MinIO要求使用的是ES 5.X系统版本。如果使用的是低版本的ES, 也没关系, ES官方支持升级迁移, 详情请看[这里](#)。

第二步：把ES集成到MinIO中

MinIO Server的配置文件默认路径是 `~/.minio/config.json`。ES配置信息是在 `notify` 这个节点下的 `elasticsearch` 节点下, 在这里为你的ES实例创建配置信息键值对, `key`是你的ES的名称, `value`是下面表格中列列的键值对集合。

参数	类型	描述
<code>enable</code>	<code>bool</code>	(必须) 是否启用这个配置?
<code>format</code>	<code>string</code>	(必须) 是 <code>namespace</code> 还是 <code>access</code>
<code>url</code>	<code>string</code>	(必须) ES地址, 比如: <code>http://localhost:9200</code>
<code>index</code>	<code>string</code>	(必须) 给MinIO用的index

以下是ES的一个配置示例:

```
"elasticsearch": {
    "1": {
        "enable": true,
        "format": "namespace",
        "url": "http://127.0.0.1:9200",
        "index": "minio_events"
    }
},
```

更新完配置文件后，重启MinIO Server让配置生效。如果一切顺利，MinIO Server会在启动时输出一行信息，类似 `SQS ARNs: arn:minio:sqs:us-east-1:1:elasticsearch`。

注意一下，你又可以再一次听从你内心的想法，想配几个ES服务就配几个，只要每个ES服务实例有不同的ID (比如前面示例中的“1”) 和配置信息。

第三步：使用MinIO客户端启用bucket通知

我们现在可以在一个叫 `images` 的存储桶上开启事件通知。一旦有文件被创建或者覆盖，一个新的ES的document会被创建或者更新到之前咱配的index里。如果一个已经存在的对象被删除，这个对应的document也会从index中删除。因此，这个 ES index里的行，就映射着 `images` 存储桶里的对象。

要配置这种存储桶通知，我们需要用到前面步骤MinIO输出的ARN信息。更多有关ARN的资料，请参考[这里](#)。

有了 `mc` 这个工具，这些配置信息很容易就能添加上。假设咱们的MinIO服务别名叫 `myminio`，可执行下列脚本：

```
mc mb myminio/images
mc event add myminio/images arn:minio:sqs:us-east-1:1:elasticsearch --suffix .jpg
mc event list myminio/images
arn:minio:sqs:us-east-1:1:elasticsearch s3:ObjectCreated:* s3:ObjectRemoved:* Filter: suffix=".jpg"
```

第四步：验证ES

上传一张图片到 `images` 存储桶。

```
mc cp myphoto.jpg myminio/images
```

使用curl查到 `minio_events` index中的内容。

```
$ curl "http://localhost:9200/minio_events/_search?pretty=true"
{
  "took" : 40,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "minio_events",
        "_type" : "event",
        "_id" : "images/myphoto.jpg",
        "_score" : 1.0,
        "_source" : {
```

```
"Records" : [
    {
        "eventVersion" : "2.0",
        "eventSource" : "minio:s3",
        "awsRegion" : "us-east-1",
        "eventTime" : "2017-03-30T08:00:41Z",
        "eventName" : "s3:ObjectCreated:Put",
        "userIdentity" : {
            "principalId" : "minio"
        },
        "requestParameters" : {
            "sourceIPAddress" : "127.0.0.1:38062"
        },
        "responseElements" : {
            "x-amz-request-id" : "14B09A09703FC47B",
            "x-minio-origin-endpoint" : "http://192.168.86.115:9000"
        },
        "s3" : {
            "s3SchemaVersion" : "1.0",
            "configurationId" : "Config",
            "bucket" : {
                "name" : "images",
                "ownerIdentity" : {
                    "principalId" : "minio"
                },
                "arn" : "arn:aws:s3:::images"
            },
            "object" : {
                "key" : "myphoto.jpg",
                "size" : 6474,
                "eTag" : "a3410f4f8788b510d6f19c5067e60a90",
                "sequencer" : "14B09A09703FC47B"
            }
        },
        "source" : {
            "host" : "127.0.0.1",
            "port" : "38062",
            "userAgent" : "MinIO (linux; amd64) minio-go/2.0.3 mc/2017-02-15T17:57:25Z"
        }
    }
]
```

这个输出显示在ES中为这个事件创建了一个document。

这里我们可以看到这个document ID就是存储桶和对象的名称。如果用的是 access 格式，这个document ID就是由ES随机生成的。

使用 Redis 发布 MinIO 事件

安装 Redis。为了演示，我们将数据库密码设为"yoursecret"。

这种通知目标支持两种格式: *namespace* 和 *access*。

如果用的是`namespace`格式，MinIO将存储桶里的对象同步成Redis hash中的条目。对于每一个条目，对应一个存储桶里的对象，其key都被设为“存储桶名称/对象名称”，value都是一个有关这个MinIO对象的JSON格式的事件数据。如果对象更新或者删除，hash中对象的条目也会相应的更新或者删除。

如果使用的是`access`，MinIO使用RPUSH将事件添加到list中。这个list中每一个元素都是一个JSON格式的list，这个list中又有两个元素，第一个元素是时间戳的字符串，第二个元素是一个含有在这个存储桶上进行操作的事件数据的JSON对象。在这种格式下，list中的元素不会更新或者删除。

下面的步骤展示如何在`namespace`和`access`格式下使用通知目标。

第一步：集成Redis到MinIO

MinIO Server的配置文件默认路径是`~/.minio/config.json`。Redis配置信息是在`notify`这个节点下的`redis`节点下，在这里为你的Redis实例创建配置信息键值对，`key`是你的Redis端的名称，`value`是下面表格中的键值对里面值的集合。

参数	类型	描述
<code>enable</code>	<code>bool</code>	(必须) 这个配置是否可用？
<code>format</code>	<code>string</code>	(必须) 是 <code>namespace</code> 还是 <code>access</code>
<code>address</code>	<code>string</code>	(必须) Redis服务地址，比如： <code>localhost:6379</code>
<code>password</code>	<code>string</code>	(可选) Redis服务密码
<code>key</code>	<code>string</code>	(必须) 事件要存储到redis key的名称。如果用的是 <code>namespace</code> 格式的话，则是一个hash，如果是 <code>access</code> 格式的话，则是一个list

下面是一个Redis配置示例：

```
"redis": {  
    "1": {  
        "enable": true,  
        "address": "127.0.0.1:6379",  
        "password": "yoursecret",  
        "key": "bucketevents"  
    }  
}
```

更新完配置文件后，重启MinIO Server让配置生效。如果一切顺利，MinIO Server会在启动时输出一行信息，类似`SQS ARNs: arn:minio:sqs:us-east-1:1:redis`。

注意一下，你永远都可以听从你内心的想法，想配几个Redis服务就配几个，只要每个Redis服务实例有不同的ID（比如前面示例中的“1”）和配置信息。

第二步：使用MinIO客户端启用bucket通知

我们现在可以在一个叫`images`的存储桶上开启事件通知。一旦有文件被创建或者覆盖，一个新的key会被创建，或者一个已经存在的key就会被更新到之前配置好的redis hash里。如果一个已经存在的对象被删除，这个对应的key也会从hash中删除。因此，这个Redis hash里的行，就映射着`images`存储桶里的`.jpg`对象。

要配置这种存储桶通知，我们需要用到前面步骤MinIO输出的ARN信息。更多有关ARN的资料，请参考[这里](#)。

有了`mc`这个工具，这些配置信息很容易就能添加上。假设咱们的MinIO服务别名叫`myminio`，可执行下列脚本：

```
mc mb myminio/images  
mc event add myminio/images arn:minio:sqs:us-east-1:1:redis --suffix .jpg  
mc event list myminio/images  
arn:minio:sqs:us-east-1:1:redis s3:ObjectCreated:* s3:ObjectRemoved:* Filter: suffix=".jpg"
```

第三步：验证Redis

启动 `redis-cli` 这个Redis客户端程序来检查Redis中的内容。运行 `monitor` Redis命令将会输出在Redis上执行的每个命令的。

```
redis-cli -a yoursecret
127.0.0.1:6379> monitor
OK
```

打开一个新的terminal终端并上传一张JPEG图片到 `images` 存储桶。

```
mc cp myphoto.jpg myminio/images
```

在上一个终端中，你将看到MinIO在Redis上执行的操作：

```
127.0.0.1:6379> monitor
OK
1490686879.650649 [0 172.17.0.1:44710] "PING"
1490686879.651061 [0 172.17.0.1:44710] "HSET" "minio_events" "images/myphoto.jpg" "{\"Records\":[{\\"eventVersion\":\"2.0\",\\\"eventSource\\\":\\\"minio:s3\\\",\\\"awsRegion\\\":\\\"us-east-1\\\",\\\"eventTime\\\":\\\"2017-03-28T07:41:19Z\\\",\\\"eventName\\\":\\\"s3:ObjectCreated:Put\\\",\\\"userIdentity\\\":{\\\"principalId\\\":\\\"minio\\\"},\\\"requestParameters\\\":{\\\"sourceIPAddress\\\":\\\"127.0.0.1:52234\\\"},\\\"responseElements\\\":{\\\"x-amz-request-id\\\":\\\"14AFFBD1ACE5F632\\\",\\\"x-minio-origin-endpoint\\\":\\\"http://192.168.86.115:9000\\\"},\\\"s3\\\":{\\\"s3SchemaVersion\\\":\\\"1.0\\\",\\\"configurationId\\\":\\\"Config\\\",\\\"bucket\\\":{\\\"name\\\":\\\"images\\\",\\\"ownerIdentity\\\":{\\\"principalId\\\":\\\"minio\\\"},\\\"arn\\\":\\\"arn:aws:s3:::images\\\"},\\\"object\\\":{\\\"key\\\":\\\"myphoto.jpg\\\",\\\"size\\\":2586,\\\"eTag\\\":\\\"5d284463f9da279f060f0ea4d11af098\\\",\\\"sequencer\\\":\\\"14AFFBD1ACE5F632\\\"},\\\"source\\\":{\\\"host\\\":\\\"127.0.0.1\\\",\\\"port\\\":\\\"52234\\\",\\\"userAgent\\\":\\\"MinIO (linux; amd64) minio-go/2.0.3 mc/2017-02-15T17:57:25Z\\\"}}}]}"
```

在这我们可以看到MinIO在 `minio_events` 这个key上执行了 `HSET` 命令。

如果用的是 `access` 格式，那么 `minio_events` 就是一个list，MinIO就会调用 `R PUSH` 添加到list中。这个list的消费者会使用 `BLPOP` 从list的最左端删除list元素。

使用NATS发布MinIO事件

安装 [NATS](#)。

第一步：集成NATS到MinIO

MinIO Server的配置文件默认路径是 `~/.minio/config.json`。参考下面的示例修改NATS的配置：

```
"nats": {
    "1": {
        "enable": true,
        "address": "0.0.0.0:4222",
        "subject": "bucketevents",
        "username": "yourusername",
        "password": "yoursecret",
        "token": "",
        "secure": false,
        "pingInterval": 0
        "streaming": {
            "enable": false,
            "clusterID": "",
            "clientID": "",
            "async": false,
            "maxPubAcksInflight": 0
        }
    }
}
```

```
        }
    },
},
```

更新完配置文件后，重启MinIO Server让配置生效。`bucketevents` 是NATS在这个例子中使用的主题。

MinIO服务也支持 [NATS Streaming mode](#)，这种模式额外提供了像 `Message/event persistence`，`At-least-once-delivery`，以及 `Publisher rate limiting` 这样的功能。如果想让MinIO服务发送通知到NATS Streaming server，参考下面示面进行配置：

```
"nats": {
    "1": {
        "enable": true,
        "address": "0.0.0.0:4222",
        "subject": "bucketevents",
        "username": "yourusername",
        "password": "yoursecret",
        "token": "",
        "secure": false,
        "pingInterval": 0
        "streaming": {
            "enable": true,
            "clusterID": "test-cluster",
            "clientID": "minio-client",
            "async": true,
            "maxPubAcksInflight": 10
        }
    }
},
```

更多关于 `clusterID`，`clientID` 的信息，请看 [NATS documentation](#)。关于 `maxPubAcksInflight`，请看 [这里](#)。

第二步：使用MinIO客户端启用bucket通知

我们现在可以在一个叫 `images` 的存储桶上开启事件通知，一旦 `myminio server`上有文件从 `images` 存储桶里删除或者上传到存储桶中，事件即被触发。在这里，ARN的值是 `arn:minio:sqs:us-east-1:1:nats`。更多有关ARN的资料，请参考[这里](#)。

```
mc mb myminio/images
mc event add myminio/images arn:minio:sqs:us-east-1:1:nats --suffix .jpg
mc event list myminio/images
arn:minio:sqs:us-east-1:1:nats s3:ObjectCreated:* s3:ObjectRemoved:* Filter: suffix=".jpg"
```

第三步：验证NATS

如果你用的是NATS server，请查看下面的示例程序来记录添加到NATS的存储桶通知。

```
package main

// Import Go and NATS packages
import (
    "log"
    "runtime"

    "github.com/nats-io/nats.go"
)
```

```

func main() {

    // Create server connection
    natsConnection, _ := nats.Connect("nats://yourusername:yoursecret@localhost:4222")
    log.Println("Connected")

    // Subscribe to subject
    log.Printf("Subscribing to subject 'bucketevents'\n")
    natsConnection.Subscribe("bucketevents", func(msg *nats.Msg) {

        // Handle the message
        log.Printf("Received message '%s\n", string(msg.Data)+"")

    })

    // Keep the connection alive
    runtime.Goexit()
}

go run nats.go
2016/10/12 06:39:18 Connected
2016/10/12 06:39:18 Subscribing to subject 'bucketevents'

```

打开一个新的terminal终端并上传一张JPEG图片到 `images` 存储桶。

```
mc cp myphoto.jpg myminio/images
```

`nats.go` 示例程序将事件通知打印到控制台。

```

go run nats.go
2016/10/12 06:51:26 Connected
2016/10/12 06:51:26 Subscribing to subject 'bucketevents'
2016/10/12 06:51:33 Received message '{"EventType":"s3:ObjectCreated:Put","Key":"images/myphoto.jpg","Records": [{"eventVersion":"2.0","eventSource":"aws:s3","awsRegion":"us-east-1","eventTime":"2016-10-12T13:51:33Z","eventName":"s3:ObjectCreated:Put","userIdentity":{"principalId":"minio"},"requestParameters":{"sourceIPAddress":"[::1]:57106"}, "responseElements":{},"s3":{"s3SchemaVersion":"1.0","configurationId":"Config","bucket":{"name":"images","ownerIdentity":{"principalId":"minio"},"arn":"arn:aws:s3:::images"},"object":{"key":"myphoto.jpg","size":56060,"eTag":"1d97bf45ecb37f7a7b699418070df08f","sequencer":"147CCD1AE054BF0D0"}}], "level":"info","msg":"","time":"2016-10-12T06:51:33-07:00"}}

```

如果你用的是NATS Streaming server,请查看下面的示例程序来记录添加到NATS的存储桶通知。

```

package main

// Import Go and NATS packages
import (
    "fmt"
    "runtime"

    "github.com/nats-io/stan.go"
)

func main() {
    natsConnection, _ := stan.Connect("test-cluster", "test-client")
    log.Println("Connected")

    // Subscribe to subject
    log.Printf("Subscribing to subject 'bucketevents'\n")
    natsConnection.Subscribe("bucketevents", func(m *stan.Msg) {

```

```

    // Handle the message
    fmt.Printf("Received a message: %s\n", string(m.Data))
}

// Keep the connection alive
runtime.Goexit()
}

go run nats.go
2017/07/07 11:47:40 Connected
2017/07/07 11:47:40 Subscribing to subject 'bucketevents'

```

打开一个新的terminal终端并上传一张JPEG图片到 `images` 存储桶。

```
ymc cp myphoto.jpg myminio/images
```

`nats.go` 示例程序将事件通知打印到控制台。

```

Received a message: {"EventType":"s3:ObjectCreated:Put","Key":"images/myphoto.jpg","Records":[{"eventVersion": "2.0","eventSource":"minio:s3","awsRegion":"","eventTime":"2017-07-07T18:46:37Z","eventName":"s3:ObjectCreated:Put","userIdentity":{"principalId":"minio"},"requestParameters":{"sourceIPAddress":"192.168.1.80:55328"},"responseElements":{"x-amz-request-id":"14CF20BD1EFD5B93","x-minio-origin-endpoint":"http://127.0.0.1:9000"},"s3":{"s3SchemaVersion":"1.0","configurationId":"Config","bucket":{"name":"images","ownerIdentity":{"principalId":"minio"},"arn":"arn:aws:s3:::images"},"object":{"key":"myphoto.jpg","size":248682,"eTag":"f1671feacb8bbf7b0397c6e9364e8c92","contentType":"image/jpeg","userDefined":{"content-type":"image/jpeg"},"versionId":"1","sequencer":"14CF20BD1EFD5B93"},"source":{"host": "192.168.1.80","port": "55328","userAgent": "MinIO (linux; amd64) minio-go/2.0.4 mc/DEVELOPMENT.GOGET"}}],"source": {"host": "192.168.1.80", "port": "55328", "userAgent": "MinIO (linux; amd64) minio-go/2.0.4 mc/DEVELOPMENT.GOGET"}}],"level": "info", "msg": "", "time": "2017-07-07T11:46:37-07:00"}

```

使用PostgreSQL发布MinIO事件

安装 PostgreSQL 数据库。为了演示，我们将"postgres"用户的密码设为 `password`，并且创建了一个 `minio_events` 数据库来存储事件信息。

这个通知目标支持两种格式: `namespace` and `access`。

如果使用的是`namespace`格式，MinIO将存储桶里的对象同步成数据库表中的行。每一行有两列: `key`和`value`。`key`是这个对象的存储桶名字加上对象名，`value`都是一个有关这个MinIO对象的JSON格式的事件数据。如果对象更新或者删除，表中相应的行也会相应的更新或者删除。

如果使用的是`access`,MinIO将将事件添加到表里，行有两列: `event_time` 和 `event_data`。`event_time`是事件在MinIO server里发生的时间，`event_data`是有关这个MinIO对象的JSON格式的事件数据。在这种格式下，不会有行会被删除或者修改。

下面的步骤展示的是如何在 `namespace` 格式下使用通知目标，`_access_` 差不多，不再赘述，我相信你可以触类旁通，举一反三，不要让我失望哦。

第一步：确保确保至少满足第低要求

MinIO要求PostgreSQL9.5版本及以上。MinIO用了PostgreSQL9.5引入的 `INSERT ON CONFLICT` (aka UPSERT) 特性,以及9.4引入的 `JSONB` 数据类型。

第二步：集成PostgreSQL到MinIO

MinIO Server的配置文件默认路径是 `~/.minio/config.json`。PostgreSQL配置信息是在 `notify` 这个节点下的 `postgresql` 节点下，在这里为你的PostgreSQL实例创建配置信息键值对，`key`是你的PostgreSQL的名称，`value`是下面表格中列列的键值对集合。

参数	类型	描述
enable	bool	(必须)此配置是否启用
format	string	(必须)是 namespace 还是 access
connectionString	string	(可选) PostgreSQL的连接参数。比如可以用来设置 sslmode
table	string	(必须)事件对应的表名，如果该表不存在，MinIO server会在启动时创建。
host	string	(可选) PostgreSQL的主机名，默认是 localhost
port	string	(可选) PostgreSQL的端口号，默认是 5432
user	string	(可选)数据库用户名，默认是运行MinIO server进程的用户
password	string	(可选)数据库密码
database	string	(可选)库名

下面是一个PostgreSQL配置示例:

```
"postgresql": {
    "1": {
        "enable": true,
        "format": "namespace",
        "connectionString": "sslmode=disable",
        "table": "bucketevents",
        "host": "127.0.0.1",
        "port": "5432",
        "user": "postgres",
        "password": "password",
        "database": "minio_events"
    }
}
```

注意一下，为了演示，咱们这把SSL禁掉了，但是为了安全起见，不建议在生产环境这么弄。

更新完配置文件后，重启MinIO Server让配置生效。如果一切顺利，MinIO Server会在启动时输出一行信息，类似 `SQS ARNs: arn:minio:sqs:us-east-1:1:postgresql`。

和之前描述的一样，你也可以添加多个PostgreSQL实例，只要ID不重复就行。

第三步：使用MinIO客户端启用bucket通知

我们现在可以在一个叫 `images` 的存储桶上开启事件通知，一旦上有文件上传到存储桶中，PostgreSQL中会insert一条新的记录或者一条已经存在的记录会被update，如果一个存在对象被删除，一条对应的记录也会从PostgreSQL表中删除。因此，PostgreSQL表中的行，对应的就是存储桶里的一个对象。

要配置这种存储桶通知，我们需要用到前面步骤中MinIO输出的ARN信息。更多有关ARN的资料，请参考[这里](#)。

有了 `mc` 这个工具，这些配置信息很容易就能添加上。假设MinIO服务别名叫 `myminio`，可执行下列脚本：

```
# Create bucket named `images` in myminio
mc mb myminio/images
# Add notification configuration on the `images` bucket using the MySQL ARN. The --suffix argument filters events.
mc event add myminio/images arn:minio:sqs:us-east-1:1:postgresql --suffix .jpg
# Print out the notification configuration on the `images` bucket.
mc event list myminio/images
mc event list myminio/images
arn:minio:sqs:us-east-1:1:postgresql s3:ObjectCreated:* ,s3:ObjectRemoved:* Filter: suffix=".jpg"
```

第四步：验证PostgreSQL

打开一个新的terminal终端并上传一张JPEG图片到 `images` 存储桶。

```
mc cp myphoto.jpg myminio/images
```

打开一个PostgreSQL终端列出表 `bucketevents` 中所有的记录。

使用MySQL发布MinIO事件

安装 MySQL。为了演示，我们将“postgres”用户的密码设为 `password`，并且创建了一个 `minio_db` 数据库来存储事件信息。这个通知目标支持两种格式: `namespace and access`。

如果使用的是`namespace`格式，MinIO将存储桶里的对象同步成数据库表中的行。每一行有两列：`key_name`和`value`。`key_name`是这个对象的存储桶名字加上对象名，`value`都是一个有关这个MinIO对象的JSON格式的事件数据。如果对象更新或者删除，表中相应的行也会相应的更新或者删除。

如果使用的是access,MinIO将将事件添加到表里，行有两列：event_time 和 event_data。event_time是事件在MinIO server里发生的时间，event_data是有关这个MinIO对象的JSON格式的事件数据。在这种格式下，不会有行会被删除或者修改。

下面的步骤展示的是如何在 `namespace` 格式下使用通知目标，`_access_` 差不多，不再赘述。

第一步：确保至少满足第低要求

MinIO要求MySQL 版本 5.7.8及以上，MinIO使用了MySQL5.7.8版本引入的 [JSON](#) 数据类型。我们使用的是MySQL5.7.17 进行的测试。

第二步：集成MySQL到MinIO

MinIO Server的配置文件默认路径是 `~/.minio/config.json`。MySQL配置信息是在 `notify` 这个节点下的 `mysql` 节点下，在这里为你的MySQL实例创建配置信息键值对，`key`是你的PostgreSQL的名称，`value`是下面表格中列列的键值对集合。

参数	类型	描述
<code>enable</code>	<code>bool</code>	(必须)此配置是否启用?
<code>format</code>	<code>string</code>	(必须)是 <code>namespace</code> 还是 <code>access</code>

<code>dsnString</code>	<code>string</code>	(可选)MySQL的 Data-Source-Name连接串。如果没设值，连接信息将使用下列参数： <code>host</code> , <code>port</code> , <code>user</code> , <code>password</code> 以及 <code>database</code>
<code>table</code>	<code>string</code>	(必须)事件对应的表名，如果该表不存在，MinIO server会在启动时创建。
<code>host</code>	<code>string</code>	MySQL server主机名(如果 <code>dsnString</code> 是空才会使用此配置)。
<code>port</code>	<code>string</code>	MySQL server端口号(如果 <code>dsnString</code> 是空才会使用此配置)。
<code>user</code>	<code>string</code>	数据库用户名(如果 <code>dsnString</code> 是空才会使用此配置)。
<code>password</code>	<code>string</code>	数据库密码(如果 <code>dsnString</code> 是空才会使用此配置)。
<code>database</code>	<code>string</code>	数据库名(如果 <code>dsnString</code> 是空才会使用此配置)。

下面是一个MySQL配置示例：

```
"mysql": {
  "1": {
    "enable": true,
    "dsnString": "",
    "table": "minio_images",
    "host": "172.17.0.1",
    "port": "3306",
    "user": "root",
    "password": "password",
    "database": "miniodb"
  }
}
```

更新完配置文件后，重启MinIO Server让配置生效。如果一切顺利，MinIO Server会在启动时输出一行信息，类似 `ARNs: arn:minio:sqs:us-east-1:1:mysql`。

和之前描述的一样，你也可以添加多个MySQL实例，只要ID不重复就行。

第三步：使用MinIO客户端启用bucket通知

我们现在可以在一个叫 `images` 的存储桶上开启事件通知，一旦上有文件上传到存储桶中，MySQL中会insert一条新的记录或者一条已经存在的记录会被update，如果一个存在对象被删除，一条对应的记录也会从MySQL表中删除。因此，MySQL表中的行，对应的就是存储桶里的一个对象。

要配置这种存储桶通知，我们需要用到前面步骤MinIO输出的ARN信息。更多有关ARN的资料，请参考[这里](#)。

有了 `mc` 这个工具，这些配置信息很容易就能添加上。假设咱们的MinIO服务别名叫 `myminio`，可执行下列脚本：

```
# Create bucket named `images` in myminio
mc mb myminio/images
# Add notification configuration on the `images` bucket using the MySQL ARN. The --suffix argument filters events.
mc event add myminio/images arn:minio:sqs:us-east-1:1:postgresql --suffix .jpg
# Print out the notification configuration on the `images` bucket.
mc event list myminio/images
arn:minio:sqs:us-east-1:1:postgresql s3:ObjectCreated:* s3:ObjectRemoved:* Filter: suffix=".jpg"
```

第四步：验证MySQL

打开一个新的terminal终端并上传一张JPEG图片到 `images` 存储桶。

```
mc cp myphoto.jpg myminio/images
```

打开一个MySQL终端列出表 `minio_images` 中所有的记录。

```

$ mysql -h 172.17.0.1 -P 3306 -u root -p miniodb
mysql> select * from minio_images;
+-----+
| key_name      | value
|               |
+-----+
| images/myphoto.jpg | {"Records": [{"s3": {"bucket": {"arn": "arn:aws:s3:::images", "name": "images", "ownerIdentity": {"principalId": "minio"}}, "object": {"key": "myphoto.jpg", "eTag": "467886be95c8ecfd71a2900e3f461b4f", "size": 26, "sequencer": "14AC59476F809FD3"}, "configurationId": "Config", "s3SchemaVersion": "1.0"}, "awsRegion": "us-east-1", "eventName": "s3:ObjectCreated:Put", "eventTime": "2017-03-16T11:29:00Z", "eventSource": "aws:s3", "eventVersion": "2.0", "userIdentity": {"principalId": "minio"}, "responseElements": {"x-amz-request-id": "14AC59476F809FD3", "x-minio-origin-endpoint": "http://192.168.86.110:9000"}, "requestParameters": {"sourceIPAddress": "127.0.0.1:38260"}}]}
+-----+
|               |
+-----+
| 1 row in set (0.01 sec)

```

使用Kafka发布MinIO事件

安装 [Apache Kafka](#).

第一步：确保确保至少满足第低要求

MinIO要求Kafka版本0.10或者0.9. MinIO内部使用了 [Shopify/sarama](#) 库，因此需要和该库有同样的版本兼容性。

第二步：集成Kafka到MinIO

MinIO Server的配置文件默认路径是 `~/minio/config.json`。参考下面的示例更新Kafka配置：

```

"kafka": {
  "1": {
    "enable": true,
    "brokers": ["localhost:9092"],
    "topic": "bucketevents"
  }
}

```

重启MinIO server让配置生效。`bucketevents` 是本示例用到的Kafka主题（topic）。

第三步：使用MinIO客户端启用bucket通知

我们现在可以在一个叫 `images` 的存储桶上开启事件通知，一旦上有文件上传到存储桶中，事件将被触发。在这里，ARN的值是 `arn:minio:sqs:us-east-1:1:kafka`。更多有关ARN的资料，请参考[这里](#)。

```
mc mb myminio/images
mc event add myminio/images arn:minio:sqs:us-east-1:1:kafka --suffix .jpg
mc event list myminio/images
arn:minio:sqs:us-east-1:1:kafka s3:ObjectCreated:* ,s3:ObjectRemoved:* Filter: suffix=".jpg"
```

第四步：验证Kafka

我们使用 `kafkacat` 将所有的通知输出到控制台。

```
kafkacat -C -b localhost:9092 -t bucketevents
```

打开一个新的terminal终端并上传一张JPEG图片到 `images` 存储桶。

```
mc cp myphoto.jpg myminio/images
```

`kafkacat` 输出事件通知到控制台。

```
kafkacat -b localhost:9092 -t bucketevents
{"EventType": "s3:ObjectCreated:Put", "Key": "images/myphoto.jpg", "Records": [{"eventVersion": "2.0", "eventSource": "aws:s3", "awsRegion": "us-east-1", "eventTime": "2017-01-31T10:01:51Z", "eventName": "s3:ObjectCreated:Put", "userIdentity": {"principalId": "88QR09S7IOT4X1IBAQ9B"}, "requestParameters": {"sourceIPAddress": "192.173.5.2:57904"}, "responseElements": {"x-amz-request-id": "149ED2FD25589220", "x-minio-origin-endpoint": "http://192.173.5.2:9000"}, "s3": {"s3SchemaVersion": "1.0", "configurationId": "Config", "bucket": {"name": "images", "ownerIdentity": {"principalId": "88QR09S7IOT4X1IBAQ9B"}, "arn": "arn:aws:s3:::images"}, "object": {"key": "myphoto.jpg", "size": 541596, "eTag": "04451d05b4faf4d62f3d538156115e2a", "sequencer": "149ED2FD25589220"}}], "level": "info", "msg": "", "time": "2017-01-31T15:31:51+05:30"}
```

使用Webhook发布MinIO事件

[Webhooks](#) 采用推的方式获取数据，而不是一直去拉取。

第一步：集成MySQL到MinIO

MinIO Server的配置文件默认路径是 `~/.minio/config.json`。参考下面的示例更新Webhook配置：

```
"webhook": {
    "1": {
        "enable": true,
        "endpoint": "http://localhost:3000/"
    }
}
```

`endpoint`是监听webhook通知的服务。保存配置文件并重启MinIO服务让配置生效。注意一下，在重启MinIO时，这个`endpoint`必须是启动并且可访问到。

第二步：使用MinIO客户端启用bucket通知

我们现在可以在一个叫 `images` 的存储桶上开启事件通知，一旦上有文件上传到存储桶中，事件将被触发。在这里，ARN的值是 `arn:minio:sqs:us-east-1:1:webhook`。更多有关ARN的资料，请参考[这里](#)。

```
mc mb myminio/images
mc mb myminio/images-thumbnail
mc event add myminio/images arn:minio:sqs:us-east-1:1:webhook --events put --suffix .jpg
```

验证事件通知是否配置正确：

```
mc event list myminio/images
```

你应该可以收到如下的响应：

```
arn:minio:sqs:us-east-1:1:webhook    s3:ObjectCreated:*    Filter: suffix=".jpg"
```

第三步：采用**Thumbnailer**进行验证

我们使用 [Thumbnailer](#) 来监听MinIO通知。如果有文件上传于是MinIO服务，Thumbnailer监听到该通知，生成一个缩略图并上传到MinIO服务。安装Thumbnailer：

```
git clone https://github.com/minio/thumbnailer/
npm install
```

然后打开Thumbnailer的 `config/webhook.json` 配置文件，添加有关MinIO server的配置，使用下面的方式启动Thumbnailer：

```
NODE_ENV=webhook node thumbnail-webhook.js
```

Thumbnailer运行在 `http://localhost:3000/`。下一步，配置MinIO server,让其发送消息到这个URL（第一步提到的），并使用 `mc` 来设置存储桶通知（第二步提到的）。然后上传一张图片到MinIO server:

```
mc cp ~/images.jpg myminio/images
.../images.jpg: 8.31 KB / 8.31 KB |████████████████████████████| 100.00% 59.42 KB/s 0s
```

稍等片刻，然后使用`mc ls`检查存储桶的内容 -，你将看到有个缩略图出现了。

```
mc ls myminio/images-thumbnail
[2017-02-08 11:39:40 IST] 992B images-thumbnail.jpg
```

注意 如果你用的是 [distributed MinIO](#),请修改所有节点的 `~/.minio/config.json`。

Minio服务限制/租户

纠删码 (多块硬盘 / 服务)

项目	参数
最大驱动器数量	Unlimited
最小驱动器数量	Unlimited
读仲裁	N / 2
写仲裁	N / 2+1

浏览器访问

项目	参数
Web浏览器上传大小限制	5GB

Limits of S3 API

项目	参数
最大桶数	无限额
每桶最大对象数	无限额
最大对象大小	5 TB
最小对象大小	0 B
每次PUT操作的最大对象大小	5 GB
每次上传的最大Part数量	10,000
Part大小	5MB到5GB. 最后一个part可以从0B到5GB
每次list parts请求可返回的part最大数量	1000
每次list objects请求可返回的object最大数量	1000
每次list multipart uploads请求可返回的multipart uploads最大数量	1000

我们认为下列AWS S3的API有些冗余或者说用处不大，因此我们在minio中没有实现这些接口。如果您有不同意见，欢迎在[github](#)上提issue。

Minio不支持的Amazon S3 Bucket API

- BucketACL (可以用 [bucket policies](#))
- BucketCORS (所有HTTP方法的所有存储桶都默认启用CORS)
- BucketLifecycle (Minio纠删码不需要)
- BucketReplication (可以用 [mc mirror](#))
- BucketVersions, BucketVersioning (可以用 [s3git](#))
- BucketWebsite (可以用 [caddy](#) or [nginx](#))
- BucketAnalytics, BucketMetrics, BucketLogging (可以用 [bucket notification APIs](#))
- BucketRequestPayment
- BucketTagging

Minio不支持的Amazon S3 Object API.

- ObjectACL (可以用 [bucket policies](#))
- ObjectTorrent

MinIO Server config.json (v18) 指南

MinIO server在默认情况下会将所有配置信息存到 `~/.minio/config.json` 文件中。以下部分提供每个字段的详细说明以及如何自定义它们。一个完整的 `config.json` 在 [这里](#)

配置目录

默认的配置目录是 `~/.minio`，你可以使用 `--config-dir` 命令行选项重写之。MinIO server在首次启动时会生成一个新的 `config.json`，里面带有自动生成的访问凭据。

```
minio server --config-dir /etc/minio /data
```

证书目录

TLS证书存在 `~/.minio/certs` 目录下，你需要将证书放在该目录下来启用 `HTTPS`。如果你是一个乐学上进的好青年，这里有一本免费的秘籍传授一你: [如何使用TLS安全的访问minio](#).

以下是一个带来TLS证书的MinIO server的目录结构。

```
$ tree ~/.minio
/home/user1/.minio
├── certs
│   ├── CAs
│   ├── private.key
│   └── public.crt
└── config.json
```

配置参数

版本

参数	类型	描述
<code>version</code>	<code>string</code>	<code>version</code> 决定了配置文件的格式，任何老版本都会在启动时自动迁移到新版本中。[请勿手动修改]

凭据

参数	类型	描述
<code>credential</code>		对象存储和Web访问的验证凭据。
<code>credential.accessKey</code>	<code>string</code>	<code>Access key</code> 长度最小是5个字符，你可以通过 <code>MINIO_ACCESS_KEY</code> 环境变量进行修改
<code>credential.secretKey</code>	<code>string</code>	<code>Secret key</code> 长度最小是8个字符，你可以通过 <code>MINIO_SECRET_KEY</code> 环境变量进行修改

示例:

```
export MINIO_ACCESS_KEY=admin
export MINIO_SECRET_KEY=password
minio server /data
```

区域 (Region)

参数	类型	描述
region	string	region 描述的是服务器的物理位置， 默认是 us-east-1 (美国东区1), 这也是亚马逊S3 的默认区域。你可以通过 MINIO_REGION_NAME 环境变量进行修改。如果不了解这块，建议不要随意修改

示例：

```
export MINIO_REGION_NAME="中国华北一区"
minio server /data
```

浏览器

参数	类型	描述
browser	string	开启或关闭浏览器访问， 默认是开启的， 你可以通过 MINIO_BROWSER 环境变量进行修改

示例：

```
export MINIO_BROWSER=off
minio server /data
```

通知

参数	类型	描述
notify		通知通过以下方式开启存储桶事件通知， 用于lambda计算
notify.amqp		通过AMQP发布MinIO事件
notify.mqtt		通过MQTT发布MinIO事件
notify.elasticsearch		通过Elasticsearch发布MinIO事件
notify.redis		通过Redis发布MinIO事件
notify.nats		通过NATS发布MinIO事件
notify.postgresql		通过PostgreSQL发布MinIO事件
notify.kafka		通过Apache Kafka发布MinIO事件
notify.webhook		通过Webhooks发布MinIO事件

了解更多

- [MinIO Quickstart Guide](#)

MinIO多租户（Multi-tenant）部署指南

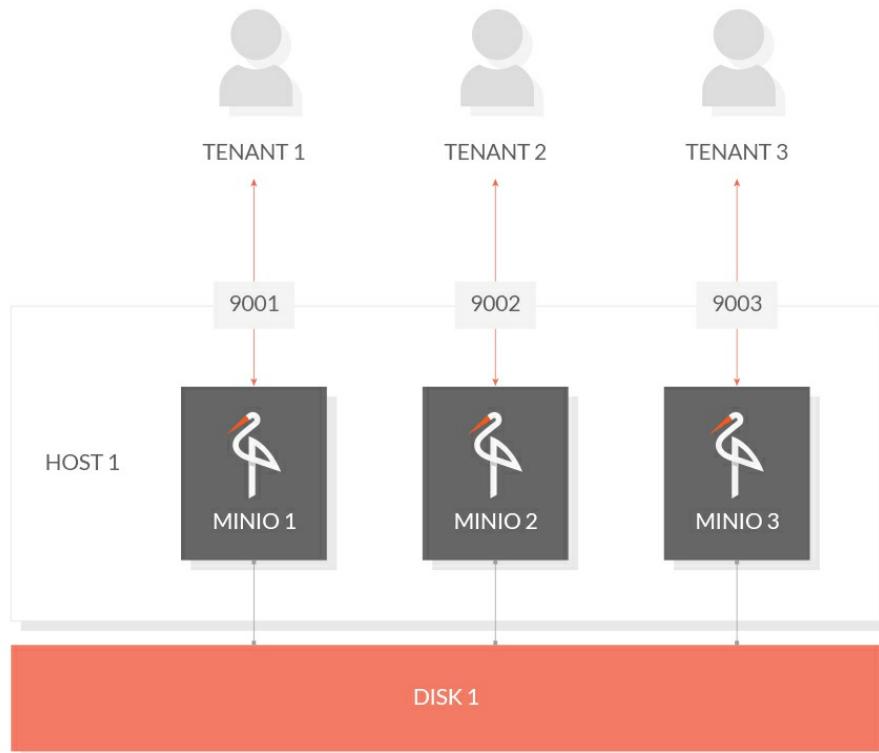
单机部署

要在单台机器上托管多个租户，为每个租户运行一个MinIO server, 使用不同的HTTPS端口、配置和数据目录。

示例1：单主机，单磁盘

以下示例在一块磁盘上托管三个租户。

```
minio --config-dir ~/tenant1 server --address :9001 /data/tenant1
minio --config-dir ~/tenant2 server --address :9002 /data/tenant2
minio --config-dir ~/tenant3 server --address :9003 /data/tenant3
```



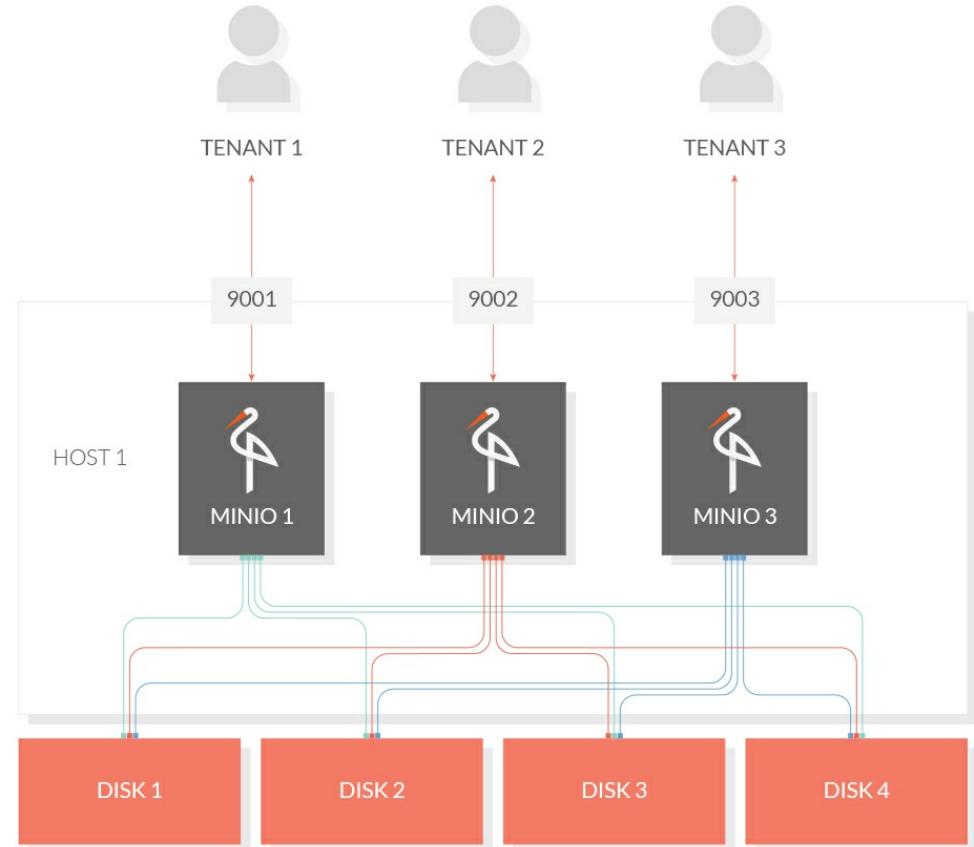
Example 1: 3 tenants on single host, single drive

示例2：单主机，多块磁盘 (**erasure code**)

以下示例在多块磁盘上托管三个租户。

```
minio --config-dir ~/tenant1 server --address :9001 /disk1/data/tenant1 /disk2/data/tenant1 /disk3/data/tenant1 /disk4/data/tenant1
minio --config-dir ~/tenant2 server --address :9002 /disk1/data/tenant2 /disk2/data/tenant2 /disk3/data/tenant2 /disk4/data/tenant2
```

```
minio --config-dir ~/tenant3 server --address :9003 /disk1/data/tenant3 /disk2/data/tenant3 /disk3/data/tenant3 /disk4/data/tenant3
```



Example 2: 3 tenants on single host, 4 drives (erasure code)

分布式部署

要在分布式环境中托管多个租户，同时运行多个分布式MinIO实例。

示例3：多主机，多块磁盘 (erasure code)

以下示例在一个4节点集群中托管三个租户。在4个节点里都执行下列命令：

```
export MINIO_ACCESS_KEY=<TENANT1_ACCESS_KEY>
export MINIO_SECRET_KEY=<TENANT1_SECRET_KEY>
minio --config-dir ~/tenant1 server --address :9001 http://192.168.10.11/data/tenant1 http://192.168.10.12/data/tenant1 http://192.168.10.13/data/tenant1 http://192.168.10.14/data/tenant1

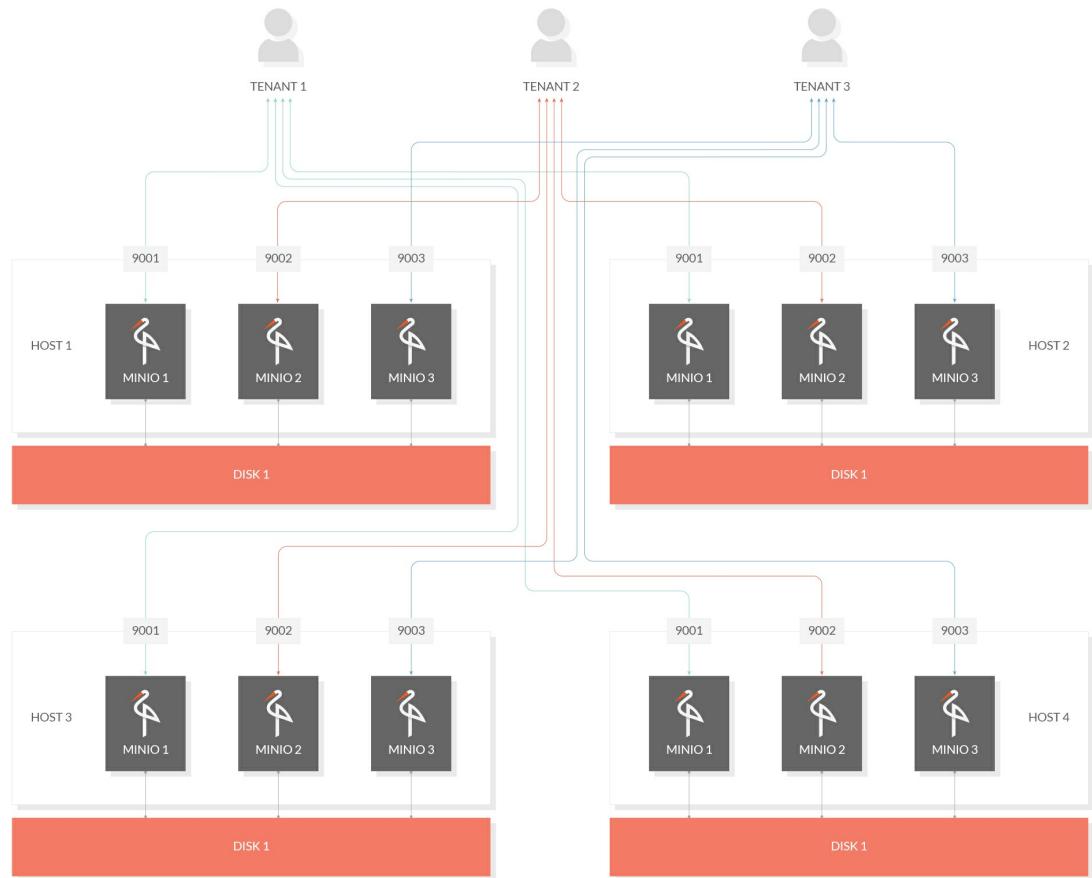
export MINIO_ACCESS_KEY=<TENANT2_ACCESS_KEY>
export MINIO_SECRET_KEY=<TENANT2_SECRET_KEY>
minio --config-dir ~/tenant2 server --address :9002 http://192.168.10.11/data/tenant2 http://192.168.10.12/data/tenant2 http://192.168.10.13/data/tenant2 http://192.168.10.14/data/tenant2

export MINIO_ACCESS_KEY=<TENANT3_ACCESS_KEY>
```

```

export MINIO_SECRET_KEY=<TENANT3_SECRET_KEY>
minio --config-dir ~/tenant3 server --address :9003 http://192.168.10.11/data/tenant3 http://192.168.10.12/data/tenant3
ta/tenant3 http://192.168.10.13/data/tenant3 http://192.168.10.14/data/tenant3

```



Example 3: 4 node distributed setup

云端可伸缩部署

对于大型多租户 MinIO 部署，我们建议使用一个流行的容器编排平台，比如 Kubernetes、DC/OS，或者是 Docker Swarm。参考 [这个文档](#)，学习如何在编排平台中使用 MinIO。

MinIO Azure 网关

MinIO网关将亚马逊S3兼容性添加到微软Azure Blob存储。

运行支持微软Azure Blob存储的MinIO网关

使用Docker

```
docker run -p 9000:9000 --name azure-s3 \
-e "MINIO_ACCESS_KEY=azureaccountname" \
-e "MINIO_SECRET_KEY=azureaccountkey" \
minio/minio gateway azure
```

使用二进制

```
export MINIO_ACCESS_KEY=azureaccountname
export MINIO_SECRET_KEY=azureaccountkey
minio gateway azure
```

使用MinIO浏览器验证

MinIO Gateway配有嵌入式网络对象浏览器。将您的Web浏览器指向<http://127.0.0.1:9000>确保您的服务器已成功启动。

使用MinIO客户端 mc 验证

mc 提供了诸如ls, cat, cp, mirror, diff等UNIX命令的替代方案。它支持文件系统和Amazon S3兼容的云存储服务。

配置 mc

```
mc config host add myazure http://gateway-ip:9000 azureaccountname azureaccountkey
```

列出微软Azure上的容器

```
mc ls myazure
[2017-02-22 01:50:43 PST]      0B ferenginar/
[2017-02-26 21:43:51 PST]      0B my-container/
[2017-02-26 22:10:11 PST]      0B test-container1/
```

了解更多

- [mc 命令行接口](#)
- [aws 命令行接口](#)
- [minio-go Go SDK](#)

MinIO GCS 网关

MinIO GCS网关将亚马逊S3兼容性添加到Google云存储。

运行支持**GCS**的**MinIO** 网关

为**GCS**创建服务帐户密钥，并获取凭据文件

1. 访问 [API控制台凭证页面](#).
2. 选择您的项目或创建一个新项目，记下你的项目ID。
3. 在凭据页面，选择 **Create credentials** 下拉项，然后选择 **Service account key**。
4. 从 **Service account**下拉项，选择 **New service account**
5. 填写 **Service account name** 和 **Service account ID**
6. 对于 **Role**, 点击下拉项，选择 **Storage -> Storage Admin** (完全控制GCS资源)
7. 点击 **Create** 按钮，下载凭据文件到你的桌面，文件名咱们就叫 `credentials.json`

注意: 设置 *Application Default Credentials*的替代方案 在 [这里](#)进行了描述。

使用 Docker

```
docker run -p 9000:9000 --name gcs-s3 \
-v /path/to/credentials.json:/credentials.json \
-e "GOOGLE_APPLICATION_CREDENTIALS=/credentials.json" \
-e "MINIO_ACCESS_KEY=minioaccountname" \
-e "MINIO_SECRET_KEY=minioaccountkey" \
minio/minio gateway gcs yourprojectid
```

使用二进制

```
export GOOGLE_APPLICATION_CREDENTIALS=/path/to/credentials.json
export MINIO_ACCESS_KEY=minioaccesskey
export MINIO_SECRET_KEY=miniosecretkey
minio gateway gcs yourprojectid
```

使用**MinIO Browser**验证

MinIO Gateway配有嵌入式网络对象浏览器。将您的Web浏览器指向<http://127.0.0.1:9000>确保您的服务器已成功启动。

使用**MinIO**客户端 `mc` 验证

`mc` 提供了诸如`ls`, `cat`, `cp`, `mirror`, `diff`等UNIX命令的替代方案。它支持文件系统和Amazon S3兼容的云存储服务。

配置 `mc`

```
mc config host add mygcs http://gateway-ip:9000 minioaccesskey miniosecretkey
```

列出**GCS**上的容器

```
mc ls mygcs
```

```
[2017-02-22 01:50:43 PST]    0B ferenginar/
[2017-02-26 21:43:51 PST]    0B my-container/
[2017-02-26 22:10:11 PST]    0B test-container1/
```

了解更多

- [mc 命令行接口](#)
- [aws 命令行接口](#)
- [minio-go Go SDK](#)

MinIO NAS网关

MinIO网关使用NAS存储支持Amazon S3。你可以在同一个共享NAS卷上运行多个minio实例，作为一个分布式的对象网关。

为NAS存储运行MinIO网关

使用Docker

```
docker run -p 9000:9000 --name nas-s3 \
-e "MINIO_ACCESS_KEY=minio" \
-e "MINIO_SECRET_KEY=minio123" \
minio/minio gateway nas /shared/nasvol
```

使用二进制

```
export MINIO_ACCESS_KEY=minioaccesskey
export MINIO_SECRET_KEY=miniosecretkey
minio gateway nas /shared/nasvol
```

使用浏览器进行验证

使用你的浏览器访问 `http://127.0.0.1:9000` ,如果能访问，恭喜你，启动成功了。

使用 mc 进行验证

`mc` 为ls, cat, cp, mirror, diff, find等UNIX命令提供了一种替代方案。它支持文件系统和兼容Amazon S3的云存储服务（AWS Signature v2和v4）。

设置 mc

```
mc config host add mynas http://gateway-ip:9000 access_key secret_key
```

列举nas上的存储桶

```
mc ls mynas
[2017-02-22 01:50:43 PST]      0B ferenginar/
[2017-02-26 21:43:51 PST]      0B my-bucket/
[2017-02-26 22:10:11 PST]      0B test-bucket1/
```

了解更多

- [mc 快速入门](#)
- [使用 aws-cli](#)
- [使用 minio-go SDK](#)

MinIO S3网关

MinIO S3 Gateway向AWS S3或任何其他与AWS S3兼容的服务中添加了MinIO功能，如MinIO浏览器和磁盘缓存。

运行适用于AWS S3的MinIO Gateway

作为运行MinIO S3网关的先决条件，默认情况下，您需要有效的AWS S3访问密钥和秘密密钥。（可选）当您通过环境变量（即AWS_ACCESS_KEY_ID）旋转AWS IAM凭证或AWS凭证时，还可以设置自定义访问/秘密密钥。

使用Docker

```
docker run -p 9000:9000 --name minio-s3 \
-e "MINIO_ACCESS_KEY=aws_s3_access_key" \
-e "MINIO_SECRET_KEY=aws_s3_secret_key" \
minio/minio gateway s3
```

使用二进制

```
export MINIO_ACCESS_KEY=aws_s3_access_key
export MINIO_SECRET_KEY=aws_s3_secret_key
minio gateway s3
```

在EC2中使用Binary

使用适用于AWS S3的IAM旋转凭证

```
export MINIO_ACCESS_KEY=custom_access_key
export MINIO_SECRET_KEY=custom_secret_key
minio gateway s3
```

如果您的后端URL是AWS S3，则MinIO网关将按以下顺序自动查找证书样式列表。

- AWS环境变量（即AWS_ACCESS_KEY_ID）
- AWS凭证文件（即AWS_SHARED_CREDENTIALS_FILE或~/ .aws / credentials）
- 基于IAM配置文件的凭据。（执行对预定义端点的HTTP调用，仅在已配置的ec2实例内部有效）

如果您希望使用AWS凭证提供受限访问权限，则需要最低权限，请确保为您的AWS用户或角色遵循以下IAM策略。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "readonly",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3>ListBucket"
            ],
            "Resource": "arn:aws:s3:::testbucket"
        },
        {
            "Sid": "readonly",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::testbucket/*"
        }
    ]
}
```

```
        "Action": "s3:HeadBucket",
        "Resource": "arn:aws:s3:::testbucket"
    }
]
}
```

运行适用于AWS S3兼容服务的MinIO Gateway

作为在AWS S3兼容服务上运行MinIO S3网关的先决条件，您需要有效的访问密钥，秘密密钥和服务端点。

使用双重加密运行MinIO Gateway

到S3的MinIO网关支持静态数据加密。支持三种类型的加密模式

- 可以将加密设置 `pass-through` 为后端
- `single encryption` (在网关)
- `double encryption` (在网关处进行单一加密，然后传递到后端)。

可以通过设置`MINIO_GATEWAY_SSE`环境变量来指定。如果未设置`MINIO_GATEWAY_SSE`和KMS，则所有加密标头都将传递到后端。如果设置了KMS环境变量，`single encryption` 则会在网关上自动执行，并将加密的对象保存在后端。

要指定 `double encryption`，对于`sse-s3`，需要将`MINIO_GATEWAY_SSE`环境变量设置为“`s3`”，而对于`sse-c`加密则需要设置为“`c`”。可以设置多个加密选项，以“;”分隔。对象在网关处被加密，并且网关也对后端进行传递。请注意，在使用SSE-C加密的情况下，网关会使用密钥派生功能（KDF）从SSE-C客户端密钥派生唯一的SSE-C密钥进行传递。

```
export MINIO_GATEWAY_SSE="s3;c"
export MINIO_KMS_VAULT_APPROLE_ID=9b56cc08-8258-45d5-24a3-679876769126
export MINIO_KMS_VAULT_APPROLE_SECRET=4e30c52f-13e4-a6f5-0763-d50e8cb4321f
export MINIO_KMS_VAULT_ENDPOINT=https://vault-endpoint-ip:8200
export MINIO_KMS_VAULT_KEY_NAME=my-minio-key
export MINIO_KMS_VAULT_AUTH_TYPE=approle
minio gateway s3
```

使用Docker

```
docker run -p 9000:9000 --name minio-s3 \
-e "MINIO_ACCESS_KEY=access_key" \
-e "MINIO_SECRET_KEY=secret_key" \
minio/minio gateway s3 https://s3_compatible_service_endpoint:port
```

使用二进制

```
export MINIO_ACCESS_KEY=access_key
export MINIO_SECRET_KEY=secret_key
minio gateway s3 https://s3_compatible_service_endpoint:port
```

MinIO 缓存

MinIO边缘缓存允许将内容存储在离应用程序更近的地方。经常访问的对象存储在基于本地磁盘的缓存中。利用MinIO网关功能进行边缘缓存

- 显着提高了任何对象到第一个字节的时间。
- 避免 S3 数据传输费用.

请参考 [本文档](#) 以开始使用MinIO Caching。

MinIO浏览器

MinIO Gateway带有基于嵌入式Web的对象浏览器。将您的Web浏览器指向<http://127.0.0.1:9000>，以确保服务器已成功启动。

借助MinIO S3网关，您可以使用MinIO浏览器浏览基于AWS S3的对象。

已知限制

- 不支持存储桶通知API。

进一步探索

- [mc 命令行界面](#)
- [aws 命令行界面](#)
- [minio-go 转到SDK](#)

MinIO HDFS网关

MinIO HDFS网关将Amazon S3 API支持添加到Hadoop HDFS文件系统中。应用程序可以同时使用S3和文件API，而无需任何数据迁移。由于网关是无状态且无共享的，因此您可以弹性地分配所需数量的MinIO实例以分配负载。

运行MinIO Gateway进行HDFS存储

使用二进制

通过 `core-site.xml` 自动从hadoop环境变量 `$HADOOP_HOME` 中读取来获取Namenode信息

```
export MINIO_ACCESS_KEY=minio
export MINIO_SECRET_KEY=minio123
minio gateway hdfs
```

您还可以覆盖namenode端点，如下所示。

```
export MINIO_ACCESS_KEY=minio
export MINIO_SECRET_KEY=minio123
minio gateway hdfs hdfs://namenode:8200
```

使用 Docker

使用docker是实验性的，大多数Hadoop环境未进行docker化，可能需要其他步骤才能使其正常工作。在这种情况下，最好只使用二进制文件。

```
docker run -p 9000:9000 \
--name hdfs-s3 \
-e "MINIO_ACCESS_KEY=minio" \
-e "MINIO_SECRET_KEY=minio123" \
minio/minio gateway hdfs hdfs://namenode:8200
```

使用MinIO浏览器进行测试

MinIO网关 带有基于Web的嵌入式对象浏览器。将您的Web浏览器指向<http://127.0.0.1:9000>，以确保服务器已成功启动。

使用MinIO Client进行测试 `mc`

`mc` 提供了诸如`ls`, `cat`, `cp`, `mirror`, `diff`等UNIX命令的现代替代方案。它支持文件系统和与Amazon S3兼容的云存储服务。

配置 `mc`

```
mc config host add myhdfs http://gateway-ip:9000 access_key secret_key
```

在HDFS上列出存储桶

```
mc ls myhdfs
[2017-02-22 01:50:43 PST]      0B user/
[2017-02-26 21:43:51 PST]      0B datasets/
```

[2017-02-26 22:10:11 PST] 0B assets/

已知限制

网关继承了HDFS存储层的以下限制：

- 没有存储桶策略支持（HDFS没有这样的概念）
- 不支持存储桶通知API（HDFS不支持fsnotify）
- 不支持服务器端加密（有意未实现）
- 不支持服务器端压缩（有意未实现）

路线图

- 对PutObject操作的其他元数据支持
- 多部分操作的其他元数据支持
- 后台附加为多部分操作提供并发支持

如果您希望解决这些问题，请打开GitHub问题。<https://github.com/minio/minio/issues>

进一步探索

- `mc` 命令行界面
- `aws` 命令行界面
- `minio-go` 转到 SDK

磁盘缓存快速入门

这里的磁盘缓存功能是指使用缓存磁盘来存储租户常用的一些数据。例如，假设你通过 `gateway azure` 设置访问一个对象并下载下来进行缓存，那接下来的请求都会直接访问缓存磁盘上的对象，直至其过期失效。此功能允许Minio用户：

- 对象的读取速度性能最佳。
- 任何对象的首字节时间得到显著改善。

开始

1. 前期条件

安装Minio - [Minio快速入门](#)。

2. 运行Minio缓存

磁盘缓存可以通过修改Minio服务的 `cache` 配置来进行开启。配置 `cache` 设置需要指定磁盘路径、缓存过期时间（以天为单位）以及使用统配符方式指定的不需要进行缓存的对象。

```
"cache": {  
    "drives": ["/mnt/drive1", "/mnt/drive2", "/mnt/drive3"],  
    "expiry": 90,  
    "exclude": ["*.pdf", "mybucket/*"]  
},
```

缓存设置也可以通过环境变量设置。设置后，环境变量会覆盖任何 `cache` 配置中的值。下面示例使用 `/mnt/drive1`, `/mnt/drive2` 和 `/mnt/drive3` 来做缓存，90天失效，并且 `mybucket` 下的所有对象以及后缀名为 `.pdf` 的对象不做缓存。

```
export MINIO_CACHE_DRIVES="/mnt/drive1,/mnt/drive2,/mnt/drive3"  
export MINIO_CACHE_EXPIRY=90  
export MINIO_CACHE_EXCLUDE="*.pdf,mybucket/*"  
minio server /export{1...24}
```

3. 验证设置是否成功

要验证是否部署成功，你可以通过浏览器或者 `mc` 来访问刚刚部署的Minio服务。你应该可以看到上传的文件在所有Minio节点上都可以访问。

了解更多

- [磁盘缓存设计](#)
- [mc快速入门](#)
- [使用 aws-cli](#)
- [使用 s3cmd](#)
- [使用 minio-go SDK](#)
- [Minio文档](#)

MinIO 监控指南

MinIO 服务器通过端点公开监视数据。监视工具可以从这些端点中选择数据。本文档列出了监视端点和相关文档。

健康检查探针

MinIO 服务器具有两个与运行状况检查相关的未经身份验证的端点，一个活动性探针（指示服务器是否工作正常），一个就绪性探针（指示服务器是否由于重负载而未接受连接）。

- 可在以下位置获得活力探针 `/minio/health/live`
- 可在以下位置获得就绪探针 `/minio/health/ready`

在[MinIO healthcheck 指南](#)中阅读有关如何使用这些端点的更多信息。

Prometheus 探测

MinIO 服务器在单个端点上公开与Prometheus兼容的数据。默认情况下，对端点进行身份验证。

- Prometheus 数据可在 `/minio/prometheus/metrics`

要使用此端点，请设置Prometheus以从该端点抓取数据。在[如何使用Prometheus监视MinIO服务器](#)中阅读有关如何配置和使用Prometheus监视MinIO服务器的更多信息。

如何使用Prometheus监控MinIO服务器

Prometheus Prometheus是最初在SoundCloud上构建的云原生监视平台。Prometheus提供了多维数据模型，其中包含通过度量标准名称和键/值对标识的时间序列数据。数据收集通过HTTP / HTTPS上的拉模型进行。通过服务发现或静态配置发现要提取数据的目标。

MinIO默认情况下将Prometheus兼容数据作为授权端点导出/minio/prometheus/metrics。希望监视其MinIO实例的用户可以指向Prometheus配置，以从该终结点抓取数据。

本文档说明了如何设置Prometheus并将其配置为从MinIO服务器抓取数据。

目录

- 先决条件
 - 1. 下载Prometheus
 - 2. 为Prometheus指标配置身份验证类型
 - 3. 配置Prometheus
 - 3.1 经过身份验证的Prometheus配置
 - 3.2 Public Prometheus配置
 - 4. `scrape_configs` `prometheus.yml`中的更新部分
 - 5. 启动Prometheus
- MinIO公开的指标列表

先决条件

要开始使用MinIO，请参阅[MinIO快速入门文档](#)。请按照以下步骤开始使用Prometheus进行MinIO监视。

1. 下载 Prometheus

为您的平台[下载最新版本](#)的Prometheus，然后解压缩

```
tar xvzf prometheus-* .tar.gz  
cd prometheus-*
```

Prometheus服务器是一个称为prometheus（或prometheus.exe在Microsoft Windows上）的二进制文件。运行二进制并通过--help标志以查看可用选项

```
./prometheus --help  
usage: prometheus [<flags>]  
  
The Prometheus monitoring server  
  
...
```

有关更多详细信息，请参阅[Prometheus文档](#)。

2. 为Prometheus指标配置身份验证类型

MinIO支持Prometheus `jwt` 或两种身份验证模式 `public`，默认情况下，MinIO以 `jwt mode` 运行。要允许对prometheus度量标准不进行身份验证就可以进行公共访问，请按如下所示设置环境。

```
export MINIO_PROMETHEUS_AUTH_TYPE="public"  
minio server ~/test
```

3. 配置Prometheus

3.1 经过身份验证的Prometheus配置

如果将MinIO配置为在不进行身份验证的情况下公开指标，则无需使用它mc来生成prometheus配置。您可以跳过进一步阅读并移至3.2部分。

MinIO中的Prometheus端点默认需要身份验证。Prometheus支持使用承载令牌方法对Prometheus抓取请求进行身份验证，并使用mc生成的默认Prometheus配置覆盖默认的Prometheus配置。要为别名生成Prometheus配置，请使用mc，如下所示 `mc admin prometheus generate <alias>`。

该命令将生成 `scrape_configs` `prometheus.yml` 的部分，如下所示：

```
scrape_configs:
- job_name: minio-job
  bearer_token: <secret>
  metrics_path: /minio/prometheus/metrics
  scheme: http
  static_configs:
    - targets: ['localhost:9000']
```

3.2 Public Prometheus配置

如果Prometheus端点身份验证类型设置为 `public`。遵循prometheus的配置足以开始从MinIO抓取指标数据。

```
scrape_configs:
- job_name: minio-job
  metrics_path: /minio/prometheus/metrics
  scheme: http
  static_configs:
    - targets: ['localhost:9000']
```

4. `scrape_configs` 更新部分prometheus.yml

要授权每个刮取请求，请将生成的 `scrape_configs` 部分复制并粘贴到prometheus.yml中，然后重新启动Prometheus服务。

5. 启动 Prometheus

通过运行启动（或）重新启动Prometheus服务

```
./prometheus --config.file=prometheus.yml
```

这 `prometheus.yml` 是配置文件的名称。现在，您可以在Prometheus仪表板中查看MinIO指标。默认情况下，可以从访问Prometheus仪表板 `http://localhost:9090`。

MinIO公开的指标列表

MinIO服务器在`/ minio / prometheus / metrics`端点上公开以下指标。所有这些都可以通过Prometheus仪表板进行访问。演示服务器的<https://play.min.io:9000/minio/prometheus/metrics> 中提供了暴露指标的完整列表及其定义。

这些是将在之后生效的新指标集 `RELEASE.2019-10-16*`。下面列出了此更新中的一些关键更改。-指标绑定到各个节点，并且不在群集范围内。集群中的每个节点都将公开自己的指标。-添加了涵盖s3和节点间流量统计信息的其他指标。-记录http统计信息和延迟的度量标准被标记为其各自的API（`putobject`, `getobject`等）。-磁盘使用情况指标已分发并标记到相

应的磁盘路径。

有关更多详细信息，请检查 [Migration guide for the new set of metrics](#)

指标列表及其定义如下。（注意：这里的实例是一个MinIO节点）

注意：

1. 这里的实例是一个MinIO节点。
 2. `s3 requests` 排除节点间请求。
- 标准go运行时指标的前缀 `go_`
 - 流程级别指标以开头 `process_`
 - `prometheus` 抓取以开头的指标 `promhttp_`
 - `disk_storage_used` : 磁盘使用的磁盘空间。
 - `disk_storage_available` : 磁盘上剩余的可用磁盘空间。
 - `disk_storage_total` : 磁盘上的总磁盘空间。
 - `minio_disks_offline` : 当前MinIO实例中的脱机磁盘总数。
 - `minio_disks_total` : 当前MinIO实例中的磁盘总数。
 - `s3_requests_total` : 当前MinIO实例中s3请求的总数。
 - `s3_errors_total` : 当前MinIO实例中s3请求中的错误总数。
 - `s3_requests_current` : 当前MinIO实例中活动s3请求的总数。
 - `internode_rx_bytes_total` : 当前MinIO服务器实例接收到的节点间字节总数。
 - `internode_tx_bytes_total` : 当前MinIO服务器实例发送到其他节点的字节总数。
 - `s3_rx_bytes_total` : 当前MinIO服务器实例接收的s3字节总数。
 - `s3_tx_bytes_total` : 当前MinIO服务器实例发送的s3字节总数。
 - `minio_version_info` : 具有提交ID的当前MinIO版本。
 - `s3_ttfb_seconds` : 保存请求的延迟信息的直方图。

除了上述指标外，MinIO还公开了低于模式的指标

缓存特定指标

启用了磁盘缓存的MinIO Gateway实例公开了与缓存相关的指标。

- `cache_data_served` :`cache_data_served`: 从缓存提供的总字节数。
- `cache_hits_total` :`cache_hits_total`: 缓存命中总数。
- `cache_misses_total` :`cache_misses_total`: 缓存未命中总数。

网关和缓存特定指标

MinIO Gateway实例公开与网关与云后端（S3, Azure和GCS Gateway）的通信相关的指标。

- `gateway_<gateway_type>_requests` :向云后端发出的请求总数。此度量标准具有method标识GET, HEAD, PUT和POST请求的标签。
- `gateway_<gateway_type>_bytes_sent` :发送到云后端的总字节数（在PUT和POST请求中）。
- `gateway_<gateway_type>_bytes_received` :从云后端接收的字节总数（在GET和HEAD请求中）。

请注意，这目前仅支持Azure, S3和GCS网关

MinIO自愈指标 - `self_heal_*`

MinIO 仅针对擦除代码部署公开与自我修复相关的指标。这些度量标准在网关或单节点单驱动器部署中不可用。请注意，只有在MinIO服务器上发生相关事件时，才会公开这些指标。

- `self_heal_time_since_last_activity` :自上一次自我修复相关活动以来经过的时间。
- `self_heal_objects_scanned` :在当前运行中由自愈线程扫描的对象数。重新开始自我修复运行时，它将重置。这用扫描的对象类型标记。

- `self_heal_objects_healed` : 当前运行中通过自愈线程修复的对象数。重新开始自我修复运行时，它将重置。这用扫描的对象类型标记。
- `self_heal_objects_heal_failed` : 当前运行中自愈失败的对象数。重新开始自我修复运行时，它将重置。这被标记为磁盘状态及其端点。

新指标集的迁移指南

本迁移指南适用于旧版本或之前的任何版本 `RELEASE.2019-10-23*`

MinIO磁盘级别指标 - `disk_*`

迁移包括

- ```
- `minio_total_disks` to `minio_disks_total`
- `minio_offline_disks` to `minio_disks_offline`
```

### MinIO磁盘级别指标 - `disk_storage_*`

这些指标只有一个标签。

- ```
- `disk`: Holds the disk path
```

迁移包括

- ```
- `minio_disk_storage_used_bytes` to `disk_storage_used`
- `minio_disk_storage_available_bytes` to `disk_storage_available`
- `minio_disk_storage_total_bytes` to `disk_storage_total`
```

## MinIO网络级指标

详细介绍了这些指标，以涵盖s3和节点间网络统计信息。

迁移包括

- ```
- `minio_network_sent_bytes_total` to `s3_tx_bytes_total` and `internode_tx_bytes_total`
- `minio_network_received_bytes_total` to `s3_rx_bytes_total` and `internode_rx_bytes_total`
```

添加的一些其他指标是

- ```
- `s3_requests_total`
- `s3_errors_total`
- `s3_ttfb_seconds`
```

# Federation 快速入门指南

本文档说明了如何使用 `Bucket lookup from DNS` 样式联合来配置MinIO。

## 开始使用

### 1. 先决条件

安装 MinIO - [MinIO 快速入门指南](#).

### 2. 以联合模式运行MinIO

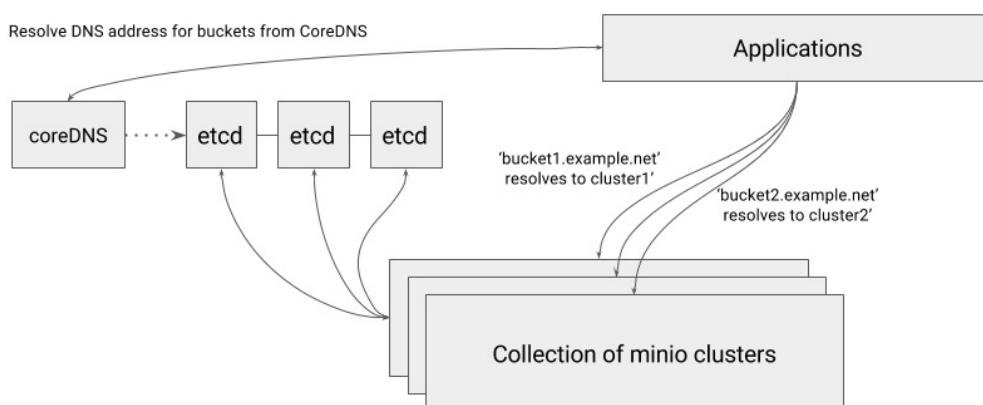
从DNS联合查找存储桶需要两个依赖项

- etcd (用于存储桶DNS服务记录)
- CoreDNS (用于基于填充的桶式DNS服务记录的DNS管理, 可选)

## 建筑

### Minio Server Federation (Bucket lookup)

Minio server to support etcd and coredns.



## 环境变量

### **MINIO\_ETCD\_ENDPOINTS**

这是您要用作MinIO联合后端的etcd服务器的逗号分隔列表。在整个联合部署中，这应该是相同的，即联合部署中的所有MinIO实例都应使用相同的 etcd后端。

### **MINIO\_DOMAIN**

这是用于联合设置的顶级域名。理想情况下，该域名应解析为 在所有联合MinIO实例之前运行的负载均衡器。域名用于创建etcd的子域条目。对于 例如，如果域名设置为 `domain.com`，水桶 `bucket1`，`bucket2` 将作为访问 `bucket1.domain.com` 和 `bucket2.domain.com`。

## MINIO\_PUBLIC\_IPS

这是用逗号分隔的IP地址列表，此MinIO实例上创建的存储桶将解析为这些IP地址。例如，可以 `bucket1` 在上访问在当前MinIO实例上创建的存储区 `bucket1.domain.com`，并且的DNS条目 `bucket1.domain.com` 将指向中设置的IP地址 `MINIO_PUBLIC_IPS`。

注意

- 对于独立和擦除代码MinIO服务器部署，此字段是必需的，以启用联合模式。
- 对于分布式部署，此字段是可选的。如果您未在联合设置中设置此字段，我们将使用传递给MinIO服务器启动的主机的IP地址，并将其用于DNS条目。`hosts passed to the MinIO server startup and use them for DNS entries.`

## 运行多个集群

集群1

```
export MINIO_ETCD_ENDPOINTS="http://remote-etcd1:2379,http://remote-etcd2:4001"
export MINIO_DOMAIN=domain.com
export MINIO_PUBLIC_IPS=44.35.2.1,44.35.2.2,44.35.2.3,44.35.2.4
minio server http://rack{1...4}.host{1...4}.domain.com/mnt/export{1...32}
```

集群2

```
export MINIO_ETCD_ENDPOINTS="http://remote-etcd1:2379,http://remote-etcd2:4001"
export MINIO_DOMAIN=domain.com
export MINIO_PUBLIC_IPS=44.35.1.1,44.35.1.2,44.35.1.3,44.35.1.4
minio server http://rack{5...8}.host{5...8}.domain.com/mnt/export{1...32}
```

在此配置中，您可以看到 `MINIO_ETCD_ENDPOINTS` 指向etcd后端的指向，该后端管理MinIO `config.json` 和存储桶DNS SRV记录。`MINIO_DOMAIN` 表示存储桶的域后缀，它将用于通过DNS解析存储桶。例如，如果您有一个诸如的存储桶 `mybucket`，则客户端现在可以使用 `mybucket.domain.com` 它直接将其自身解析为正确的集群。`MINIO_PUBLIC_IPS` 指向可以访问每个群集的公共IP地址，这对于每个群集都是唯一的。

注意：`mybucket` 仅存在于一个群集中，`cluster1` 或者 `cluster2` 这是随机的，并且由 `domain.com` 解析方式决定，如果存在循环DNS，`domain.com` 则将随机选择哪个群集可以提供存储桶。

## 3. 接口升级到 etcdv3

从发布运行MinIO联盟用户 `RELEASE.2018-06-09T03-43-35Z` 到 `RELEASE.2018-07-10T01-42-11Z`，应该ETCD服务器上现有桶数据迁移到 `etcdv3 API`，和更新版本CoreDNS向 `1.2.0` 他们MinIO服务器更新到最新版本之前。

这是为什么需要这样做的一些背景-MinIO服务器发布 `RELEASE.2018-06-09T03-43-35Z` 到 `RELEASE.2018-07-10T01-42-11Z` 使用过的`etcdv2 API`，以将存储区数据存储到etcd服务器。这是由于 `etcdv3 CoreDNS` 服务器不支持该功能。因此，即使MinIO使用 `etcdv3 API` 存储存储桶数据，CoreDNS也将无法读取并将其用作DNS记录。

现在CoreDNS [supports etcdv3](#)，MinIO服务器使用 `etcdv3 API` 将存储桶数据存储到etcd服务器。由于 `etcdv2` 和 `etcdv3 API` 不兼容，因此使用 `tcdv2 API` 无法存储使用API存储的数据 `etcdv3`。因此，在完成迁移之前，当前MinIO版本将看不到先前MinIO版本存储的存储桶数据。

CoreOS team has documented the steps required to migrate existing data from `etcdv2` to `etcdv3` in [this blog post](#). Please refer the post and migrate etcd data to `etcdv3 API`. CoreOS团队已在[this blog post](#)中记录了将现有数据从迁移 `etcdv2` 到 `etcdv3` 所需的步骤。请参考该帖子，并将etcd数据迁移到API。

## 4. 测试您的设置

要测试此设置，请通过浏览器或访问MinIO服务器 `mc`。您将看到可以从所有MinIO端点访问上载的文件。

## 进一步探索

- [mc](#) 与MinIO服务器一起使用
- [aws-cli](#) 与MinIO服务器一起使用
- [s3cmd](#) 与MinIO服务器一起使用
- [minio-go](#) SDK与MinIO Server一起使用
- [MinIO文档网站](#)

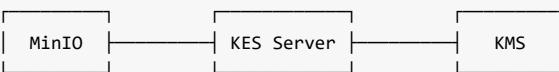
## KMS指南

MinIO使用密钥管理系统（KMS）支持SSE-S3。如果客户端请求SSE-S3，或启用了自动加密，则MinIO服务器会使用唯一的对象密钥对每个对象进行加密，该对象密钥受 KMS管理的主密钥保护。

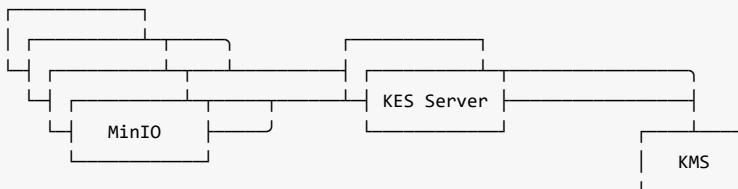
MinIO仍提供本机Hashicorp Vault支持。但是，此功能已弃用，将来可能会删除。因此，强烈建议您使用下面的体系结构和KMS指南。如果您必须维护旧版MinIO-Vault部署，则可以在此处找到旧版文档。

## 建筑与概念

KMS将MinIO作为面向应用程序的存储系统与安全密钥存储区分开，并且可以由专门的安全团队进行管理。MinIO 通过我们的KES project支持常用的KMS实现，例如 Hashicorp Vault。通过KES，可以利用存储基础架构（MinIO群集）水平扩展KMS。通常，MinIO-KMS基础结构如下所示：



当您将存储基础架构扩展到多个MinIO群集时，您的架构应如下所示：



请注意，所有MinIO群集均仅具有“其自己的” KES实例的连接，而不能直接访问Vault（作为一种可能的KMS实现）。每个KES实例将处理“其” MinIO群集发出的所有加密/解密请求，从而使中央KMS实现不必处理大量流量。相反，每个KES实例都将使用中央KMS实现作为安全密钥存储，并从中获取所需的主密钥。

## 入门指南

在随后的章节中，该指南显示了如何使用Hashicorp Vault作为KMS实施来设置MinIO-KMS部署。因此，它显示了如何设置和配置：

- Vault服务器作为中央密钥库。
- 一个KES服务器实例，作为MinIO和保险柜之间的中间件。
- MinIO实例本身。

请注意，为简便起见，本指南使用自签名证书。在生产部署中，应使用由“公共”（例如，让我们加密）或组织内部的CA颁发的X.509证书。

本指南说明如何在同一台计算机上设置三台不同的服务器：

- Vault服务器为 <https://127.0.0.1:8200>
- KES服务器为 <https://127.0.0.1:7373>
- MinIO服务器为 <https://127.0.0.1:9000>

## 1 先决条件

安装MinIO，KES和Vault。对于MinIO，请参阅 [MinIO 快速入门指南](#)。然后安装KES并下载适用于您的操作系统和平台的最新Vault二进制文件

## 2 生成TLS证书

由于KES将对象加密密钥发送给MinIO，并且Vault将主密钥（用于加密对象加密密钥）发送给KES，因此我们绝对需要MinIO、KES和Vault之间的TLS连接。因此，我们需要至少生成两个TLS证书。

### 2.1 为保险柜生成TLS证书

要为保险柜的证书生成新的私钥，请运行以下openssl命令：

```
openssl ecparam -genkey -name prime256v1 | openssl ec -out vault-tls.key
```

然后通过以下方式为私钥/公钥对生成新的TLS证书：

```
openssl req -new -x509 -days 365 \
-key vault-tls.key \
-out vault-tls.crt \
-subj "/C=/ST=/L=/O=/CN=localhost" \
-addext "subjectAltName = IP:127.0.0.1"
```

您可以忽略输出消息，例如：req：没有为主体属性C提供任何值，已跳过。OpenSSL只是告诉您尚未为证书主题指定国家/地区，州/直辖市。您可能需要调整X.509主题（-subj参数）和主题备用名称（SAN）。请注意，这是一个自签名证书。对于生产部署，此证书应由CA颁发。

### 2.2 为KES生成TLS证书

要为KES的证书生成新的私钥，请运行以下openssl命令：

```
openssl ecparam -genkey -name prime256v1 | openssl ec -out kes-tls.key
```

要为KES的证书生成新的私钥，请运行以下openssl命令：

```
openssl req -new -x509 -days 365 \
-key kes-tls.key \
-out kes-tls.crt \
-subj "/C=/ST=/L=/O=/CN=localhost" \
-addext "subjectAltName = IP:127.0.0.1"
```

您可以忽略输出消息，例如：req：没有为主体属性C提供任何值，已跳过。OpenSSL只是告诉您尚未为证书主题指定国家/地区，州/直辖市。您可能需要调整X.509主题（-subj参数）和主题备用名称（SAN）。请注意，这是一个自签名证书。对于生产部署，此证书应由CA颁发。

### 2.3 为MinIO生成TLS证书（可选）

此步骤是可选的。但是，我们建议您通过TLS来上传/下载S3对象-尤其是当它们应该在存储后端使用KMS加密时。

看有关配置MinIO和TLS的[MinIO TLS指南](#)。

## 3 设置保险柜

在类似Unix的系统上，Vault使用mlock syscall来防止操作系统将内存中的数据写入磁盘（交换）。因此，您应该赋予Vault可执行文件使用mlock syscall的能力，而无需以root用户身份运行进程。为此，请运行：

```
sudo setcap cap_ipc_lock=+ep $(readlink -f $(which vault))
```

然后创建保管库配置文件：

```

cat > vault-config.json <<EOF
{
 "api_addr": "https://127.0.0.1:8200",
 "backend": {
 "file": {
 "path": "vault/file"
 }
 },
 "default_lease_ttl": "168h",
 "max_lease_ttl": "720h",
 "listener": {
 "tcp": {
 "address": "0.0.0.0:8200",
 "tls_cert_file": "vault-tls.crt",
 "tls_key_file": "vault-tls.key",
 "tls_min_version": "tls12"
 }
 }
}
EOF

```

请注意，我们使用文件后端运行Vault。为了获得高可用性，您可能需要使用其他后端，例如[etcd](#)或[consul](#)。

最后，通过以下方式启动Vault服务器：

```
vault server -config vault-config.json
```

### 3.1 初始化和解封保管箱

在单独的终端窗口中设置环境 `VAULT_ADDR`。保险柜服务器的变量：

```
export VAULT_ADDR='https://127.0.0.1:8200'
```

此外，`export VAULT_SKIP_VERIFY=true` 如果Vault使用自签名TLS证书，则可能要运行。当Vault提供由计算机信任的CA颁发的TLS证书（例如，让我们加密）时，则无需运行此命令。

然后通过以下方式初始化保险柜：

```
vault operator init
```

保管箱将打印 `n`（默认情况下为5）解封密钥共享，其中至少 `m`（至少3）为重新生成实际解封密钥才能解封保管库。因此，请务必记住它们。特别是，请将那些未密封的密钥共享放在安全且持久的位置。

您应该看到类似于以下内容的输出：

```

Unseal Key 1: eyW/+8ZtsgT81Cb0e80VxzJAQP51Y7Dcamnze+JnWEDE
Unseal Key 2: 0tZn+7QQCxphpHwTm6/dC3LpP5JG1bY16PK8Sy79R+P2
Unseal Key 3: cmhs+AUMXUuB6Lzsvgcbp3bRT6VDGQjgCBwB2xm0ANeF
Unseal Key 4: /fTPpec5fWpGqWHK+uhnnTNMQyAb15a1Ui4iq2yNgyqj
Unseal Key 5: UPdDVPTo+H6ko+20NKmagK40M0skqOBw4y/S51WpgVy

Initial Root Token: s.zaU4Gbcu0Wh46uj2V3VuUde0

Vault is initialized with 5 key shares and a key threshold of 3. Please securely
distribute the key shares printed above. When the Vault is re-sealed,
restarted, or stopped, you must supply at least 3 of these keys to unseal it
before it can start servicing requests.

```

```
Vault does not store the generated master key. Without at least 3 key to
reconstruct the master key, Vault will remain permanently sealed!
```

```
It is possible to generate new unseal keys, provided you have a quorum of
existing unseal keys shares. See "vault operator rekey" for more information.
```

现在，设置环境。变量 `VAULT_TOKEN` 到命令之前打印的根令牌：

```
export VAULT_TOKEN=s.zaU4Gbcu0Wh46uj2V3VuUde0
```

然后，使用任何先前生成的密钥共享来打开Vault的密封。

```
vault operator unseal eyW/+8ZtsgT81Cb0e80VxzJAQP51Y7Dcamnze+JnWEDT
vault operator unseal 0tZn+7QQCxphpHwTm6/dC3LpP5JGIbY16PK8Sy79R+P2
vault operator unseal cmhs+AUMXUuB6Lzsvgcbp3bRT6VDGQjgCBwB2xm0ANeF
```

提交足够的有效密钥共享后，保管箱将被密封 并能够处理请求。

### 3.2 启用保险柜的K / V后端

加密主密钥（而不是对象加密密钥）将存储在Vault中。因此，我们需要启用Vault的K / V后端。为此，请运行：

```
vault secrets enable kv
```

### 3.3 启用AppRole身份验证

由于我们希望稍后将一个/多个KES服务器连接到Vault，因此必须启用 AppRole身份验证。为此，请运行：

```
vault auth enable approle
```

### 3.4 为K / V引擎创建访问策略

以下策略确定应用程序（即KES服务器）如何与Vault交互。

```
cat > minio-kes-policy.hcl <<EOF
path "kv/minio/*" {
 capabilities = ["create", "read", "delete"]
}

EOF
```

观察路径前缀 `minio` 在 `kv/minio/*`。此前缀确保 KES服务器只能在 `minio` /下进行读取，而只能在-下进行写入 `'some-app'`。如何在K / V引擎上分隔域取决于您的基础结构 和安全要求。

然后，我们将政策上传到保险柜：

```
vault policy write minio-key-policy ./minio-kes-policy.hcl
```

### 3.5 创建一个新的AppRole ID并将其绑定到策略

现在，我们需要创建一个新的AppRole ID并授予该ID特定的权限。该应用程序（即KES服务器）将通过AppRole角色ID 和机密ID 向Vault进行身份验证，并且仅允许执行特定策略授予的操作。

因此，我们首先为KES服务器创建一个新角色：

```
vault write auth/approle/role/kes-role token_num_uses=0 secret_id_num_uses=0 period=5m
```

然后，我们将策略绑定到角色：

```
vault write auth/approle/role/kes-role policies=minio-key-policy
```

最后，我们从Vault请求AppRole角色ID和秘密ID。一，角色ID：

```
vault read auth/approle/role/kes-role/role-id
```

然后是秘密ID：

```
vault write -f auth/approle/role/kes-role/secret-id
```

我们只对 `secret_id` 不感兴趣 `secret_id_accessor`。

## 4 设置KES

与Vault类似，KES `mlock` 在Linux系统上使用`syscall`来防止OS将内存中的数据写入磁盘（交换）。因此，您应该赋予KES可执行文件使用 `mlock syscall` 的能力，而无需以root用户身份运行进程。为此，请运行：

```
sudo setcap cap_ipc_lock=+ep $(readlink -f $(which kes))
```

### 4.1 为MinIO创建标识

连接到KES服务器（mTLS）时，每个用户或应用程序必须出示有效的X.509证书。KES服务器将接受/拒绝连接尝试，并根据证书应用策略。

因此，每个MinIO群集都需要一个X.509 TLS证书来进行客户端身份验证。您可以通过运行以下命令来创建（自签名）证书：

```
kes tool identity new MinIO --key=minio.key --cert=minio.cert --time=8760h
```

注意，`MinIO`是**subject name**。您可以为您的部署方案选择一个更合适的名称。此外，对于生产部署，我们建议获取由CA颁发的用于客户端身份验证的TLS证书。

要获取X.509证书的身份，请运行：

```
kes tool identity of minio.cert
```

此命令可与任何（有效）X.509证书一起使用-无论如何创建它-并产生类似于以下内容的输出：

```
Identity: dd46485bedc9ad2909d2e8f9017216eec4413bc5c64b236d992f7ec19c843c5f
```

### 4.2 创建KES配置文件

现在，我们可以创建KES配置文件并启动KES服务器。

```
The TCP address (ip:port) for the KES server to listen on.
address: 0.0.0.0:7373

tls:
 key: kes-tls.key
 cert: kes-tls.crt
```

```

policy:
 minio:
 paths:
 - /v1/key/create/minio-*
 - /v1/key/generate/minio-*
 - /v1/key/decrypt/minio-*
 identities:
 - dd46485bedc9ad2909d2e8f9017216eec4413bc5c64b236d992f7ec19c843c5f

 cache:
 expiry:
 any: 5m0s
 unused: 20s

 keys:
 vault:
 endpoint: https://127.0.0.1:8200 # The Vault endpoint - i.e. https://127.0.0.1:8200
 prefix: minio # The domain resp. prefix at Vault's K/V backend

 approle:
 id: "" # Your AppRole Role ID
 secret: "" # Your AppRole Secret ID
 retry: 15s # Duration until the server tries to re-authenticate after connection loss.

 tls:
 ca: vault-tls.crt # Since we use self-signed certificates

 status:
 ping: 10s

```

请 `identities` 在 `policy` 部分中将的值更改为您的身份 `minio.cert`。另外，插入您在保险柜设置过程中之前创建的AppRole角色ID和密码ID。您可以[在此处](#)找到包含所有可用参数的文档化配置文件。

最后，通过以下方式启动KES服务器：

```
kes server --config=kes-config.yaml --mlock --root=disabled --auth=off
```

请注意，由于不需要特殊的根标识，因此我们实际上将其禁用。有关KES访问控制模型和身份验证的更多信息：[KES Concepts](#)。此外，请注意，由于客户端X.509证书是自签名证书，因此我们将其禁用 `--auth=off`。

### 4.3 创建一个新的主密钥

在继续进行MinIO设置之前，我们需要创建一个新的主密钥。因此，我们使用 MinIO身份和KES CLI。

在新的终端窗口中，通过以下方式成为MinIO身份：

```
export KES_CLIENT_TLS_KEY_FILE=minio.key
export KES_CLIENT_TLS_CERT_FILE=minio.cert
```

然后运行以下命令来创建主密钥：

```
kes key create minio-key-1 -k
```

`-k` 由于我们使用自签名证书，因此仅需要该标志。另外，请注意，基于服务器配置文件，仅允许MinIO标识创建/使用以开头的主密钥 `minio-`。因此，尝试创建密钥（例如）`kes key create my-key-1 -k` 将失败，并出现策略错误禁止的消息。

## 5 设置MinIO服务器

MinIO服务器将需要知道KES服务器端点，用于身份验证和授权的mTLS客户端证书以及默认的主密钥名称。

```
export MINIO_KMS_KES_ENDPOINT=https://localhost:7373
export MINIO_KMS_KES_KEY_FILE=minio.key
export MINIO_KMS_KES_CERT_FILE=minio.cert
export MINIO_KMS_KES_KEY_NAME=minio-key-1
export MINIO_KMS_KES_CA_PATH=kes-tls.crt
```

MINIO\_KMS\_KES\_CAPATH 由于我们使用自签名证书，因此仅需要。

(可选) 启用自动加密以自动加密上传的对象：

```
export MINIO_KMS_AUTO_ENCRYPTION=on
```

有关自动加密的更多信息，请参见：[附录A](#)

然后启动MinIO服务器：

```
export MINIO_ACCESS_KEY=minio
export MINIO_SECRET_KEY=minio123
minio server ~/export
```

## 附录A-自动加密

(可选) 您可以指示MinIO服务器使用KES 服务器中的密钥自动加密所有对象-即使客户端在S3 PUT操作期间未指定任何加密标头。

当MinIO操作员希望确保存储在MinIO上的所有数据在写入存储后端之前都已加密时，自动加密特别有用。

要启用自动加密，请将环境变量设置为 `on`：

```
export MINIO_KMS_AUTO_ENCRYPTION=on
```

请注意，自动加密只会影响没有S3加密标头的请求。因此，如果S3客户端发送例如SSE-C标头，则MinIO将使用客户端发送的密钥对对象进行加密，并且不会与KMS进行联系。要验证自动加密，请使用以下 `mc` 命令：

```
mc cp test.file myminio/crypt/
test.file: 5 B / 5 B [██████████]
[██████████] 100.00% 337 B/s 0s
mc stat myminio/crypt/test.file
Name : test.file
...
Encrypted :
X-Amz-Server-Side-Encryption: AES256
```

## 附录B-指定主密钥

除了正确的KMS设置，您还可以使用KMS主密钥测试 MinIO加密。通过`env`的单个主密钥。变量仅用于测试目的，不建议用于生产部署。

KMS主密钥由一个主密钥ID（CMK）和编码为十六进制值的256位主密钥组成，并以分隔`:`。可以使用以下命令直接指定KMS主密钥：

```
export MINIO_KMS_MASTER_KEY=minio-demo-key:6368616e676520746869732070617373776f726420746f206120736563726574
```

请使用您自己的主密钥。可以使用以下命令在Linux / Mac / BSD系统上生成随机主密钥：

```
head -c 32 /dev/urandom | xxd -c 32 -ps
```

或者，您可以将主密钥作为Docker secret传递。

```
echo "my-minio-key:6368616e676520746869732070617373776f726420746f206120736563726574" | docker secret create kms_master_key
```

要使用其他秘密名称，请按照上述说明操作，并`kms_master_key`用您的自定义名称（例如`my_kms_master_key`）替换。然后，将`MINIO_KMS_MASTER_KEY_FILE`环境变量设置为您的秘密名称：

```
export MINIO_KMS_MASTER_KEY_FILE=my_kms_master_key
```

## 进一步探索

- `mc` 与MinIO服务器一起使用
- `aws-cli` 与MinIO服务器一起使用
- `s3cmd` 与MinIO服务器一起使用
- `minio-go` SDK 与MinIO Server一起使用
- MinIO文档网站

# 选择API快速入门指南

传统的对象检索始终是整个实体，即5 GiB对象的GetObject将始终返回5 GiB数据。S3 Select API允许我们使用简单的SQL表达式来检索数据的子集。通过使用Select API仅检索应用程序所需的数据，可以实现大幅的性能改进。

您可以使用Select API查询具有以下功能的对象：

- CSV, JSON和Parquet-对象必须为CSV, JSON或Parquet格式。
- UTF-8是Select API支持的唯一编码类型。
- GZIP或BZIP2-可以使用GZIP或BZIP2压缩CSV和JSON文件。Select API支持使用GZIP, Snappy, LZ4对Parquet进行行压缩。Parquet对象不支持整个对象压缩。
- 服务器端加密-Select API支持查询受服务器端加密保护的对象。

当未键入值时（例如，读取CSV数据时），将根据上下文执行类型推断和值的自动转换。如果存在，则CAST功能将覆盖自动转换。

## 1. 先决条件

- 从[这里](#)安装MinIO Server。
- 熟悉AWS S3 API。
- 熟悉Python和安装依赖项。

## 2. 安装 boto3

`aws-sdk-python` 从此处[安装](#)适用于Python的AWS开发工具包官方文档

## 3. 例子

例如，让我们以gzip压缩的CSV文件为例。如果没有S3 Select，我们将需要下载，解压缩和处理整个CSV以获得所需的数据。使用Select API，可以使用简单的SQL表达式仅从您感兴趣的CSV中返回数据，而不是检索整个对象。以下Python示例显示了如何 `Location` 从包含CSV格式数据的对象中检索第一列。

请更换 `endpoint_url` , `aws_access_key_id` , `aws_secret_access_key` , `Bucket` 和 `Key` 在这个本地设置 `select.py` 文件。

```
#!/usr/bin/env/python3
import boto3

s3 = boto3.client('s3',
 endpoint_url='http://localhost:9000',
 aws_access_key_id='minio',
 aws_secret_access_key='minio123',
 region_name='us-east-1')

r = s3.select_object_content(
 Bucket='mycsvbucket',
 Key='sampledata/TotalPopulation.csv.gz',
 ExpressionType='SQL',
 Expression="select * from s3object s where s.Location like '%United States%'",
 InputSerialization={
 'CSV': {
 "FileHeaderInfo": "USE",
 },
 'CompressionType': 'GZIP',
 },
)
```

```

 OutputSerialization={'CSV': {}},
)

 for event in r['Payload']:
 if 'Records' in event:
 records = event['Records']['Payload'].decode('utf-8')
 print(records)
 elif 'Stats' in event:
 statsDetails = event['Stats']['Details']
 print("Stats details bytesScanned: ")
 print(statsDetails['BytesScanned'])
 print("Stats details bytesProcessed: ")
 print(statsDetails['BytesProcessed'])

```

## 4. 运行程序

使用以下命令将样本数据集上传到MinIO。

```

$ curl "https://population.un.org/wpp/Download/Files/1_Indicators%20(Standard)/CSV_FILES/WPP2019_TotalPopulationBySex.csv" > TotalPopulation.csv
$ mc mb myminio/mycsvbucket
$ gzip TotalPopulation.csv
$ mc cp TotalPopulation.csv.gz myminio/mycsvbucket/sampleddata/

```

现在，让我们继续运行我们的选择示例，以查询 `Location` 匹配的内容 `United States`。

```

$ python3 select.py
840,United States of America,2,Medium,1950,1950.5,79233.218,79571.179,158804.395

840,United States of America,2,Medium,1951,1951.5,80178.933,80726.116,160905.035

840,United States of America,2,Medium,1952,1952.5,81305.206,82019.632,163324.851

840,United States of America,2,Medium,1953,1953.5,82565.875,83422.307,165988.190
....
....
...

Stats details bytesScanned:
6758866
Stats details bytesProcessed:
25786743

```

F有关更详细的SELECT SQL参考，请参见[此处](#)

## 5. 进一步探索

- `mc` 与MinIO服务器一起使用
- `mc sql` 与MinIO服务器一起使用
- `minio-go` SDK 与MinIO服务器一起使用
- `aws-cli` 与MinIO服务器一起使用
- `s3cmd` 与MinIO服务器一起使用
- [MinIO文档网站](#)

## 6. 实施状况

- 支持完整的AWS S3 [SELECT SQL](#)语法。
- 支持所有[运算符](#)。
- 支持所有聚合，条件，类型转换和字符串函数。
- `FROM S3Object[*].path` 尚未评估JSON路径表达式。
- 尚不支持大号（有符号的64位范围之外）。
- 日期的功能 `DATE_ADD` , `DATE_DIFF` , `EXTRACT` 并 `UTCNOW` 使用类型转换沿 `CAST` 的 `TIMESTAMP` 数据类型，目前支持。
- 尚未遵守AWS S3的[保留关键字](#)列表。
- The Date functions `DATE_ADD` , `DATE_DIFF` , `EXTRACT` and `UTCNOW` along with type conversion using `CAST` to the `TIMESTAMP` data type are currently supported.
- CSV输入字段（甚至带引号）也不能包含换行符，即使 `RecordDelimiter` 是其他内容也是如此。

# 压缩指南

MinIO服务器允许流式压缩以确保有效的磁盘空间使用。压缩是在飞行中发生的，即对象在写入磁盘之前已被压缩。MinIO [klauspost/compress/s2](#) 由于其稳定性和性能而使用流式压缩。

该算法专门针对机器生成的内容进行了优化。每个CPU内核的写吞吐量通常至少为300MB / s。解压缩速度通常至少为1GB / s。这意味着在原始IO低于这些数字的情况下，压缩不仅会减少磁盘使用量，而且有助于提高系统吞吐量。通常，当可以压缩内容时，在旋转磁盘系统上启用压缩将提高速度。

## 开始使用

### 1. 先决条件

安装MinIO - [MinIO 快速入门指南](#).

### 2. 通过压缩运行MinIO

可以通过更新 `compress` MinIO服务器配置的配置设置来启用压缩。配置 `compress` 设置采用扩展名和mime类型进行压缩。

```
$ mc admin config get myminio compression
compression_extensions=".txt,.log,.csv,.json,.tar,.xml,.bin" mime_types="text/*,application/json,application/xml"
```

默认配置包括最常见的高度可压缩的内容扩展名和mime类型。

```
$ mc admin config set myminio compression extensions=".pdf" mime_types="application/pdf"
```

使用默认扩展名和mime类型对所有内容启用压缩。

```
~ mc admin config set myminio compression
```

压缩设置也可以通过环境变量来设置。设置后，环境变量将覆盖``compress``服务器配置中定义的配置设置。

```
```bash
export MINIO_COMPRESS="on"
export MINIO_COMPRESS_EXTENSIONS=".pdf,.doc"
export MINIO_COMPRESS_MIME_TYPES="application/pdf"
```

3. 注意

- 已经压缩的对象不具有可压缩的模式，因此不适合进行压缩。这样的对象不能产生有效的效率 [LZ compression](#)，这是无损数据压缩的适用性。以下是不适合压缩的常见文件和内容类型的列表。

- 扩展名

```
| gz | (GZIP) | bz2 | (BZIP2) | rar | (WinRAR) | zip | (ZIP) | 7z | (7-Zip) | xz | (LZMA) | mp4 | (MP4) |
| mkv | (MKV media) | mov | (MOV)
```

- 内容类型

```
| video/* || audio/* || application/zip || application/x-gzip || application/zip || application/x-bz2 || application/x-compress || application/x-xz |
```

即使所有类型都启用了压缩，所有具有这些扩展名和mime类型的文件都将从压缩中排除。

- MinIO不支持压缩加密，因为压缩和加密在一起可能为诸如 [CRIME and BREACH](#)
- MinIO不支持网关（[Azure / GCS / NAS](#)）实现的压缩。

测试设置

要测试此设置，请练习使用 `mc` 和 `mc ls` 在数据目录上使用来对服务器进行调用，以查看对象的大小。

进一步探索

- [mc 与 MinIO 服务器一起使用](#)
- [aws-cli 与 MinIO 服务器一起使用](#)
- [s3cmd 与 MinIO 服务器一起使用](#)
- [minio-go SDK 与 MinIO 服务器一起使用](#)
- [MinIO文档网站](#)

MinIO多用户快速入门指南

除了在服务器启动期间创建的默认用户外，MinIO还支持多个长期用户。服务器启动后，可以添加新用户，并且可以将服务器配置为拒绝或允许这些用户访问存储桶和资源。本文档说明了如何添加/删除用户以及修改其访问权限。

开始使用

在本文档中，我们将详细说明如何配置多个用户。

1. 先决条件

- 安装 [mc - MinIO Client快速入门指南](#)
- 安装 [MinIO - MinIO 快速入门指南](#)
- 配置 etcd (仅在网关或联合身份验证模式下才需要) - [Etcd V3 快速入门指南](#)

2. 使用固定策略创建新用户

使用 `mc admin policy` 创建罐装政策。服务器提供罐装政策的默认设置，即 `writeonly`，`readonly` 和 `readwrite`（这些政策适用于所有服务器上的资源）。可以使用 `mc admin policy` 命令通过自定义策略来覆盖这些设置。

创建新的罐头策略文件 `getonly.json`。使用此策略，用户可以下载下的所有对象 `my-bucketname`。

```
cat > getonly.json << EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:GetObject"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::my-bucketname/*"
      ],
      "Sid": ""
    }
  ]
}
EOF
```

`getonly` 使用 `getonly.json` 策略文件按名称创建新的固定策略。

```
mc admin policy add myminio getonly getonly.json
```

`newuser` 在MinIO使用上创建一个新用户 `mc admin user`。

```
mc admin user add myminio newuser newuser123
```

成功创建用户后，您现在可以 `getonly` 为该用户应用策略。

```
mc admin policy set myminio getonly user=newuser
```

3. 创建一个新组

```
mc admin group add myminio newgroup newuser
```

成功创建组后，您现在可以 `getonly` 对该组应用策略。

```
mc admin policy set myminio getonly group=newgroup
```

4. 禁用用户

禁用用户 `newuser`。

```
mc admin user disable myminio newuser
```

禁用组 `newgroup`。

```
mc admin group disable myminio newgroup
```

5. 删除用户

删除用户 `newuser`。

```
mc admin user remove myminio newuser
```

从组中删除用户 `newuser`。

```
mc admin group remove myminio newgroup newuser
```

删除组 `newgroup`。

```
mc admin group remove myminio newgroup
```

6. 更改用户或组策略

将用户策略更改 `newuser` 为 `putonly` 固定策略。

```
mc admin policy set myminio putonly user=newuser
```

将组策略更改 `newgroup` 为 `putonly` 固定策略。

```
mc admin policy set myminio putonly group=newgroup
```

7. 列出所有用户或组

列出所有启用和禁用的用户。

```
mc admin user list myminio
```

列出所有启用或禁用的组。

```
mc admin group list myminio
```

8. 配置 mc

```
mc config host add myminio-newuser http://localhost:9000 newuser newuser123 --api s3v4  
mc cat myminio-newuser/my-bucketname/my-objectname
```

进一步探索

- [MinIO客户端完整指南](#)
- [MinIO STS快速入门指南](#)
- [MinIO管理员完成指南](#)
- [MinIO文档网站](#)

MinIO STS快速入门指南

MinIO安全令牌服务（STS）是一种终结点服务，使客户端可以请求MinIO资源的临时凭据。临时凭据的工作原理几乎与默认管理员凭据相同，但有一些区别：

- 顾名思义，临时证书是短期的。可以将它们配置为持续几分钟到几小时的时间。凭证过期后，MinIO将不再识别它们或允许使用它们发出的API请求进行任何类型的访问。
- 临时凭证不需要与应用程序一起存储，而是动态生成的，并在请求时提供给应用程序。当临时凭证（或什至之前）到期时，应用程序可以请求新凭证。

以下是使用临时凭证的优点：

- 无需在应用程序中嵌入长期凭证。
- 无需提供对存储桶和对象的访问，而无需定义静态凭据。
- 临时凭证的有效期有限，无需旋转或显式吊销它们。过期的临时凭证无法重复使用。

身份联盟

认证网	描述
Client grants	让应用程序client_grants使用任何知名第三方身份提供商（例如Keycloak，WSO2）进行请求。这被称为客户端授予方法以进行临时访问。使用这种方法可以帮助客户端保持MinIO凭据的安全。MinIO STS支持客户端授权，并针对身份提供商（例如WSO2，Keycloak）进行了测试。
WebIdentity	让用户使用任何OpenID（OIDC）兼容的Web身份提供商（例如Facebook，Google等）请求临时凭据。
AssumeRole	让MinIO用户使用用户访问权限和密钥请求临时凭证。
AD/LDAP	让AD / LDAP用户使用AD / LDAP用户名和密码来请求临时凭据。

开始使用

在本文档中，我们将详细说明如何配置所有先决条件。

注意：如果仅对AssumeRole API感兴趣，请跳到[此处](#)

1. 先决条件

- 配置 [wso2](#)
- 配置 [opa](#) (可选)
- 配置 [etcd](#) (仅在网关或联合方式下需要可选)

2. 使用 WSO2 设置 MinIO

确保已按照上一步操作并独立配置每个软件，完成后，我们现在可以继续使用MinIO STS API和MinIO服务器来使用这些凭据来执行对象API操作。

```
export MINIO_ACCESS_KEY=minio
export MINIO_SECRET_KEY=minio123
export MINIO_IDENTITY_OPENID_CONFIG_URL=https://localhost:9443/oauth2/oidcdiscovery/.well-known/openid-configuration
export MINIO_IDENTITY_OPENID_CLIENT_ID="843351d4-1080-11ea-aa20-271ecba3924a"
minio server /mnt/data
```

3. 使用 WSO2, ETCD 设置 MinIO 网关

确保已按照上一步操作并独立配置每个软件，完成后，我们现在可以继续使用MinIO STS API和MinIO网关来使用这些凭据来执行对象API操作。

注意：MinIO网关要求将etcd配置为使用STS API。

```
export MINIO_ACCESS_KEY=aws_access_key
export MINIO_SECRET_KEY=aws_secret_key
export MINIO_IDENTITY_OPENID_CONFIG_URL=https://localhost:9443/oauth2/oidcdiscovery/.well-known/openid-configuration
export MINIO_IDENTITY_OPENID_CLIENT_ID="843351d4-1080-11ea-aa20-271ecba3924a"
export MINIO_ETCD_ENDPOINTS=http://localhost:2379
minio gateway s3
```

4. 使用client-grants.go进行测试

在另一个终端上，运行 client-grants.go 一个示例客户端应用程序，该应用程序从身份提供者（在我们的情况下为WSO2）获取JWT访问令牌。使用返回的访问令牌响应，通过STS API调用从MinIO服务器获取新的临时凭据 AssumeRoleWithClientGrants 。

```
go run client-grants.go -cid PoEgXP6uV045IsENRngDXj5Au5Ya -csec eKsw6z8Ct0JVbtr0WvhRWL4TUCga

##### Credentials
{
    "accessKey": "NUIB0RZYT2HG2BMRSXR",
    "secretKey": "qQ1P507CFPc5m5IXf1vYhuVTFj7BRVJqh0FqZ86S",
    "expiration": "2018-08-21T17:10:29-07:00",
    "sessionToken": "eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJy2Nlc3NLZXkiOiJ0VUlCT1JaWVRWMkhHMkJNUlNYUiIsImF1ZCI6IlBvRwdYUDZ1Vk80NU1zRU5SbmdEWGo1QXU1WEiLCJhenAiOiJQb0VnWFA2dVZPNDVJc0VOUm5nRFhqNUF1NVlhIiwiZXhwIjoxNTM0ODk2NjI5LCJpYXQiOjE1MzQ40TMwMjksImlzcyI6Imh0dHBzOi8vbG9jYWxob3N0Ojk0NDMvb2F1dGgyL3Rva2VuIiwanRpIjoiNjY20TzjZTctN2U1Ny00ZjU5LWI0MWQtM2E1YTMzZGZiNjA4In0.eJONnVaSVHypixKEARSMnSkgr-2mlC2Sr4fEGJitLcJF_at3LeNdTHv0_oHsv6ZZA3zueVGgF1VXM1REgr9LXA"
```

进一步探索

- [MinIO管理员完成指南](#)
- [MinIO文档网站](#)

MinIO部署快速入门

MinIO是一个[云原生](#)的应用程序，旨在在多租户环境中以可持续的方式进行扩展。编排（orchestration）平台为MinIO的扩展提供了非常好的支撑。以下是各种编排平台的MinIO部署文档：

Orchestration平台
Docker Swarm
Docker Compose
Kubernetes
DC/OS

为什么说MinIO是云原生的（cloud-native）？

云原生这个词代表的是一些思想的集合，比如微服务部署，可伸缩，而不是说把一个单体应用改造成容器部署。一个云原生的应用在设计时就考虑了移植性和可伸缩性，而且可以通过简单的复制即可实现水平扩展。现在兴起的编排平台，像 Swarm、Kubernetes以及DC/OS，让大规模集群的复制和管理变得前所未有的简单，哪里不会点哪里。

容器提供了隔离的应用执行环境，编排平台通过容器管理以及复制功能提供了无缝的扩展。MinIO继承了这些，针对每个租户提供了存储环境的隔离。

MinIO是建立在云原生的基础上，有纠删码、分布式和共享存储这些特性。MinIO专注于并且只专注于存储，而且做的还不错。它可以通过编排平台复制一个MinIO实例就实现了水平扩展。

在一个云原生环境中，伸缩性不是应用的一个功能而是编排平台的功能。

现在的应用、数据库，key-store这些，很多都已经部署在容器中，并且通过编排平台进行管理。MinIO提供了一个健壮的、可伸缩、AWS S3兼容的对象存储，这是MinIO的立身之本，凭此在云原生应用中占据一席之地。

使用Docker Swarm部署MinIO

Docker Engine在Swarm模式下提供集群管理和编排功能。 MinIO服务器可以在Swarm的分布式模式下轻松部署，创建一个
多租户，高可用性和可扩展的对象存储。

从Docker Engine v1.13.0 (Docker Compose v3.0)开始，Docker Swarm和Compose 二者cross-compatible。这允许将
Compose file用作在Swarm上部署服务的模板。 我们使用Docker Compose file创建分布式MinIO设置。

1. 前提条件

- 熟悉Swarm mode key concepts.
- Docker engine v1.13.0运行在[networked host machines]集群上(<https://docs.docker.com/engine/swarm/swarm-tutorial/#/three-networked-host-machines>).

2. 创建Swarm

在管理节点上创建一个swarm，请运行下面的命令

```
docker swarm init --advertise-addr <MANAGER-IP>
```

一旦swarm初始化了，你可以看到下面的响应信息

```
docker swarm join \
--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \
192.168.99.100:2377
```

你现在可以运行上述命令添加worker节点到swarm。更多关于创建swarm的细节步骤，请访问[Docker documentation site](#).

3. 为MinIO创建Docker secret

```
echo "AKIAIOSFODNN7EXAMPLE" | docker secret create access_key -
echo "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY" | docker secret create secret_key -
```

4. 部署分布式minio服务

在你的Swarm master上下载Docker Compose file，然后运行下面的命令

```
docker stack deploy --compose-file=docker-compose-secrets.yaml minio_stack
```

这将把Compose file里描述的服务部署为Docker stack minio_stack。更多 docker stack 命令参考。

在stack成功部署之后，你可以通过MinIO Client mc 或者浏览器访问[http://\[Node_Public_IP_Address\]:\[Expose_Port_on_Host\]](http://[Node_Public_IP_Address]:[Expose_Port_on_Host])来访问你的MinIO server。

4. 删除分布式MinIO services

删除分布式MinIO services以及相关的网络，请运行下面的命令

```
docker stack rm minio_stack
```

Swarm不会自动删除为MinIO服务创建的host volumes,如果下次新的MinIO服务不熟到swarm上，可能会导致损坏。因此，我们建议手动删除所有MinIO使用的volumes。为此，到每一个swarm的节点上，列出所有的volumes

```
docker volume ls
```

然后删除 minio_stack volumes

```
docker volume rm volume_name
```

注意事项

- 默认情况下Docker Compose file使用的是最新版的MinIO server的Docker镜像，你可以修改image tag来拉取指定版本的[MinIO Docker image](#)。
 - 默认情况下会创建4个minio实例，你可以添加更多的MinIO服务（最多总共16个）到你的MinIO Comose deployment。添加一个服务
 - 复制服务定义并适当地更改新服务的名称。
 - 更新每个服务中的命令部分。
 - 更新要为新服务公开的端口号。另外，请确保分配给新服务的端口尚未使用。
- 关于分布式MinIO的更多资料，请访问[这里](#)。
- 默认情况下，MinIO服务使用的是 local volume driver. 更多配置选项，请访问[Docker documentation](#)。
 - Docker compose file中的MinIO服务使用的端口是9001到9004，这允许多个服务在主机上运行。更多配置选项，请访问[Docker documentation](#).
 - Docker Swarm默认使用的是ingress做负载均衡，你可以根据需要配置[external load balancer based](#)。

了解更多

- [Docker Swarm mode概述](#)
- [MinIO Docker快速入门](#)
- [使用Docker Compose部署MinIO](#)
- [MinIO纠删码快速入门](#)

使用Kubernetes部署MinIO

Kubernetes的部署和状态集提供了在独立，分布式或共享模式下部署MinIO服务器的完美平台。在Kubernetes上部署MinIO有多种选择，您可以选择最适合您的。

- MinIO Helm Chart通过一个简单的命令即可提供自定义而且简单的MinIO部署。更多关于MinIO Helm部署的资料，请访问[这里](#)。
- 你也可以浏览Kubernetes [MinIO示例](#)，通过 `.yaml` 文件来部署MinIO。

1. 前提条件

- 默认standalone模式下，需要开启Beta API的Kubernetes 1.4+。
- [distributed 模式](#)，需要开启Beta API的Kubernetes 1.5+。
- 底层支持PV provisioner。
- 你的K8s集群里需要有Helm package manager [installed](#)。

2. 使用Helm Chart部署MinIO

安装 MinIO chart

```
$ helm install stable/minio
```

以上命令以默认配置在Kubernetes群集上部署MinIO。以下部分列出了MinIO图表的所有可配置参数及其默认值。

配置

参数	描述	默认值
<code>image</code>	MinIO镜像名称	<code>minio/minio</code>
<code>imageTag</code>	MinIO镜像tag. 可选值在 这里 .	<code>RELEASE.2017-08-05T00-00-53Z</code>
<code>imagePullPolicy</code>	Image pull policy	<code>Always</code>
<code>mode</code>	MinIO server模式 (<code>standalone</code> , <code>shared</code> 或者 <code>distributed</code>)	<code>standalone</code>
<code>numberOfNodes</code>	节点数(仅对分布式模式生效). 可选值 <code>4 <= x <= 16</code>	<code>4</code>
<code>accessKey</code>	默认access key	<code>AKIAIOSFODNN7EXAMPLE</code>
<code>secretKey</code>	默认secret key	<code>wJalrXUtnFEMI/K7MDENG/bPxRfCYEXAMPLEKEY</code>
<code>configPath</code>	默认配置文件路径	<code>~/.minio</code>
<code>mountPath</code>	默认挂载路径	<code>/export</code>
<code>serviceType</code>	Kubernetes service type	<code>LoadBalancer</code>
<code>servicePort</code>	Kubernetes端口	<code>9000</code>
<code>persistence.enabled</code>	是否使用持久卷存储数据	<code>true</code>
<code>persistence.size</code>	持久卷大小	<code>10Gi</code>
<code>persistence.storageClass</code>	持久卷类型	<code>generic</code>
<code>persistence.accessMode</code>	ReadWriteOnce 或者 ReadOnly	<code>ReadWriteOnce</code>
<code>resources</code>	CPU/Memory 资源需求/限制	<code>Memory: 256Mi, CPU: 100m</code>

你可以通过 `--set key=value[,key=value]` 给 `helm install`。比如，

```
$ helm install --name my-release \
--set persistence.size=100Gi \
stable/minio
```

上述命令部署了一个带上100G持久卷的MinIO服务。

或者，您可以提供一个YAML文件，用于在安装chart时指定参数值。例如，

```
$ helm install --name my-release -f values.yaml stable/minio
```

分布式MinIO

默认情况下，此图表以独立模式提供MinIO服务器。要在[分布式模式](#)中配置MinIO服务器，请将 mode 字段设置为 distributed，

```
$ helm install --set mode=distributed stable/minio
```

上述命令部署了个带有4个节点的分布式MinIO服务器。要更改分布式MinIO服务器中的节点数，请设置 numberOfNodes 属性。

```
$ helm install --set mode=distributed,numberOfNodes=8 stable/minio
```

上述命令部署了个带有8个节点的分布式MinIO服务器。注意一下， numberOfNodes 取值范围是[4,16]。

StatefulSet 限制，适用于分布式MinIO

- StatefulSets需要持久化存储，所以如果 mode 设成 distributed 的话， persistence.enabled 参数不生效。
- 卸载分布式MinIO版本时，需要手动删除与StatefulSet关联的卷。

Shared MinIO

如需采用[shared mode](#)部署MinIO，将 mode 设为 shared，

```
$ helm install --set mode=shared stable/minio
```

上述命令规定了4个MinIO服务器节点，一个存储。要更改共享的MinIO部署中的节点数，请设置 numberOfNodes 字段，

```
$ helm install --set mode=shared,numberOfNodes=8 stable/minio
```

上述命令规定了MinIO服务有8个节点，采用shared模式。

持久化

这里规定了PersistentVolumeClaim并将相应的持久卷挂载到默认位置 /export。您需要Kubernetes集群中的物理存储才能使其工作。如果您宁愿使用 emptyDir，请通过以下方式禁用PersistentVolumeClaim：

```
$ helm install --set persistence.enabled=false stable/minio
```

"当Pod分配给节点时，首先创建一个emptyDir卷，只要该节点上的Pod正在运行，它就会存在。当某个Pod由于任何原因从节点中删除时，emptyDir中的数据将永久删除。"

3. 使用Helm更新MinIO版本

您可以更新现有的MinIO Helm Release以使用较新的MinIO Docker镜像。为此，请使用 `helm upgrade` 命令：

```
$ helm upgrade --set imageTag=<replace-with-minio-docker-image-tag> <helm-release-name> stable/minio
```

如果更新成功，你可以看到下面的输出信息

```
Release "your-helm-release" has been upgraded. Happy Helming!
```

4. 卸载Chart

假设你的版本被命名为 `my-release`，使用下面的命令删除它：

```
$ helm delete my-release
```

该命令删除与chart关联的所有Kubernetes组件，并删除该release。

提示

- 在Kubernetes群集中运行的chart的实例称为release。安装chart后，Helm会自动分配唯一的release名称。你也可以通过下面的命令设置你心仪的名称：

```
$ helm install --name my-release stable/minio
```

- 为了覆盖默认的秘钥，可在运行`helm install`时将access key和secret key做为参数传进去。

```
$ helm install --set accessKey=myaccesskey,secretKey=mysecretkey \
stable/minio
```

了解更多

- [MinIO纠删码快速入门](#)
- [Kubernetes文档](#)
- [Helm package manager for kubernetes](#)

使用Docker Compose部署MinIO

Docker Compose允许定义和运行单主机，多容器Docker应用程序。

使用Compose，您可以使用Compose文件来配置MinIO服务。然后，使用单个命令，您可以通过你的配置创建并启动所有分布式MinIO实例。分布式MinIO实例将部署在同一主机上的多个容器中。这是建立基于分布式MinIO的开发，测试和分期环境的好方法。

1. 前提条件

- 熟悉 [Docker Compose](#).
- Docker已经在本机安装，从[这里](#)下载相关的安装器。

2. 在Docker Compose上运行分布式MinIO

在Docker Compose上部署分布式MinIO,请下载[docker-compose.yaml](#)到你的当前工作目录。Docker Compose会pull MinIO Docker Image,所以你不需要手动去下载MinIO binary。然后运行下面的命令

GNU/Linux and macOS

```
docker-compose pull  
docker-compose up
```

Windows

```
docker-compose.exe pull  
docker-compose.exe up
```

现在每个实例都可以访问，端口从9001到9004，请在浏览器中访问<http://127.0.0.1:9001/>

注意事项

- 默认情况下Docker Compose file使用的是最新版的MinIO server的Docker镜像，你可以修改image tag来拉取指定版本的[MinIO Docker image](#).
 - 默认情况下会创建4个minio实例，你可以添加更多的MinIO服务（最多总共16个）到你的MinIO Comose deployment。添加一个服务
 - 复制服务定义并适当地更改新服务的名称。
 - 更新每个服务中的命令部分。
 - 更新要为新服务公开的端口号。另外，请确保分配给新服务的端口尚未使用。
- 关于分布式MinIO的更多资料，请访问[这里](#).
- Docker compose file中的MinIO服务使用的端口是9001到9004，这允许多个服务在主机上运行。

了解更多

- [Docker Compose概述](#)
- [MinIO Docker快速入门](#)
- [使用Docker Swarm部署MinIO](#)
- [MinIO纠删码快速入门](#)

MinIO客户端快速入门指南

MinIO Client (mc)为ls, cat, cp, mirror, diff, find等UNIX命令提供了一种替代方案。它支持文件系统和兼容Amazon S3的云存储服务（AWS Signature v2和v4）。

<code>ls</code>	列出文件和文件夹。
<code>mb</code>	创建一个存储桶或一个文件夹。
<code>cat</code>	显示文件和对象内容。
<code>pipe</code>	将一个STDIN重定向到一个对象或者文件或者STDOUT。
<code>share</code>	生成用于共享的URL。
<code>cp</code>	拷贝文件和对象。
<code>mirror</code>	给存储桶和文件夹做镜像。
<code>find</code>	基于参数查找文件。
<code>diff</code>	对两个文件夹或者存储桶比较差异。
<code>rm</code>	删除文件和对象。
<code>events</code>	管理对象通知。
<code>watch</code>	监听文件和对象的事件。
<code>policy</code>	管理访问策略。
<code>session</code>	为cp命令管理保存的会话。
<code>config</code>	管理mc配置文件。
<code>update</code>	检查软件更新。
<code>version</code>	输出版本信息。

Docker容器

稳定版

```
docker pull minio/mc
docker run minio/mc ls play
```

尝鲜版

```
docker pull minio/mc:edge
docker run minio/mc:edge ls play
```

注意：上述示例默认使用MinIO[演示环境](#)做演示，如果想用 mc 操作其它S3兼容的服务，采用下面的方式来启动容器：

```
docker run -it --entrypoint=/bin/sh minio/mc
```

然后使用 `mc config` 命令。

macOS

Homebrew

使用[Homebrew](#)安装mc。

```
brew install minio/stable/mc
mc --help
```

GNU/Linux

下载二进制文件

平台	CPU架构	URL
GNU/Linux	64-bit Intel	http://dl.minio.org.cn/client/mc/release/linux-amd64/mc

```
chmod +x mc  
./mc --help
```

Microsoft Windows

下载二进制文件

平台	CPU架构	URL
Microsoft Windows	64-bit Intel	http://dl.minio.org.cn/client/mc/release/windows-amd64/mc.exe

```
mc.exe --help
```

通过源码安装

通过源码安装仅适用于开发人员和高级用户。`mc update` 命令不支持基于源码安装的更新通知。请从<https://min.io/download/#minio-client>下载官方版本。

如果您没有Golang环境，请参照[如何安装Golang](#)。

```
go get -d github.com/minio/mc  
cd ${GOPATH}/src/github.com/minio/mc  
make
```

添加一个云存储服务

如果你打算仅在POSIX兼容文件系统中使用`mc`,那你可以直接略过本节, 跳到[日常使用](#)。

添加一个或多个S3兼容的服务, 请参考下面说明。`mc` 将所有的配置信息都存储在`~/.mc/config.json` 文件中。

```
mc config host add <ALIAS> <YOUR-S3-ENDPOINT> <YOUR-ACCESS-KEY> <YOUR-SECRET-KEY> [--api API-SIGNATURE]
```

别名就是给你的云存储服务起了一个短点的外号。S3 endpoint, access key 和 secret key 是你的云存储服务提供的。API签名是可选参数, 默认情况下, 它被设置为"S3v4"。

示例-MinIO云存储

从MinIO服务获得URL、access key和secret key。

```
mc config host add minio http://192.168.1.51 BKIKJAA5BMMU2RH06IBB V7f1CwQqAcwo80UEIJEjc5gVQUSSx5ohQ9GSrr12 --  
api s3v4
```

示例-Amazon S3云存储

参考[AWS Credentials指南](#)获取你的AccessKeyId和SecretAccessKey。

```
mc config host add s3 https://s3.amazonaws.com BKIKJAA5BMMU2RH06IBB V7f1CwQqAcwo80UEIJEjc5gVQUSSx5ohQ9GSrr12  
--api s3v4
```

示例 -Google云存储

参考[Google Credentials Guide](#)获取你的AccessKeyId和SecretAccessKey。

```
mc config host add gcs https://storage.googleapis.com BKIKJAA5BMMU2RH06IBB V8f1CwQqAcwo80UEIJEjc5gVQUSSx5ohQ  
9GSrr12 --api s3v2
```

注意：Google云存储只支持旧版签名版本V2，所以你需要选择S3v2。

验证

mc 预先配置了云存储服务URL：<https://play.min.io>，别名“play”。它是一个用于研发和测试的MinIO服务。如果想测试Amazon S3，你可以将“play”替换为“s3”。

示例：

列出<https://play.min.io>上的所有存储桶。

```
mc ls play  
[2016-03-22 19:47:48 PDT]    0B my-bucketname/  
[2016-03-22 22:01:07 PDT]    0B mytestbucket/  
[2016-03-22 20:04:39 PDT]    0B mybucketname/  
[2016-01-28 17:23:11 PST]    0B newbucket/  
[2016-03-20 09:08:36 PDT]    0B s3git-test/
```

日常使用

Shell别名

你可以添加shell别名来覆盖默认的Unix工具命令。

```
alias ls='mc ls'  
alias cp='mc cp'  
alias cat='mc cat'  
alias mkdir='mc mb'  
alias pipe='mc pipe'  
alias find='mc find'
```

Shell自动补全

你也可以下载[autocomplete/bash_autocomplete](https://raw.githubusercontent.com/minio/mc/master/autocomplete/bash_autocomplete)到`/etc/bash_completion.d/`，然后将其重命名为`mc`。别忘了在这个文件运行`source`命令让其在你的当前shell上可用。

```
sudo wget https://raw.githubusercontent.com/minio/mc/master/autocomplete/bash_autocomplete -O /etc/bash_compl  
etion.d/mc  
source /etc/bash_completion.d/mc  
mc <TAB>  
admin config diff ls mirror policy session update watch  
cat cp events mb pipe rm share version
```

了解更多

- [MinIO Client完全指南](#)
- [MinIO快速入门](#)
- [MinIO官方文档](#)

贡献

请遵守[MinIO贡献者指南](#)

MinIO Client完全指南

MinIO Client (mc)为ls, cat, cp, mirror, diff, find等UNIX命令提供了一种替代方案。它支持文件系统和兼容Amazon S3的云存储服务（AWS Signature v2和v4）。

ls	列出文件和文件夹。
mb	创建一个存储桶或一个文件夹。
cat	显示文件和对象内容。
pipe	将一个STDIN重定向到一个对象或者文件或者STDOUT。
share	生成用于共享的URL。
cp	拷贝文件和对象。
mirror	给存储桶和文件夹做镜像。
find	基于参数查找文件。
diff	对两个文件夹或者存储桶比较差异。
rm	删除文件和对象。
events	管理对象通知。
watch	监视文件和对象的事件。
policy	管理访问策略。
config	管理mc配置文件。
update	检查软件更新。
version	输出版本信息。

1. 下载MinIO Client

Docker稳定版

```
docker pull minio/mc
docker run minio/mc ls play
```

Docker尝鲜版

```
docker pull minio/mc:edge
docker run minio/mc:edge ls play
```

注意: 上述示例默认使用MinIO演示环境做演示, 如果想用 mc 操作其它S3兼容的服务, 采用下面的方式来启动容器:

```
docker run -it --entrypoint=/bin/sh minio/mc
```

然后使用 `mc config` 命令。

Homebrew (macOS)

使用Homebrew安装mc。

```
brew install minio/stable/mc
mc --help
```

下载二进制文件(GNU/Linux)

平台	CPU架构	URL
GNU/Linux	64-bit Intel	http://dl.minio.org.cn/client/mc/release/linux-amd64/mc

```
chmod +x mc  
./mc --help
```

下载二进制文件(Microsoft Windows)

平台	CPU架构	URL
Microsoft Windows	64-bit Intel	http://dl.minio.org.cn/client/mc/release/windows-amd64/mc.exe

```
mc.exe --help
```

通过源码安装

通过源码安装仅适用于开发人员和高级用户。`mc update` 命令不支持基于源码安装的更新通知。请从[minio-client](#)下载官方版本。

如果您没有Golang环境，请按照[如何安装Golang](#)。

```
go get -d github.com/minio/mc  
cd ${GOPATH}/src/github.com/minio/mc  
make
```

2. 运行MinIO Client

GNU/Linux

```
chmod +x mc  
./mc --help
```

macOS

```
chmod 755 mc  
./mc --help
```

Microsoft Windows

```
mc.exe --help
```

3. 添加一个云存储服务

如果你打算仅在POSIX兼容文件系统中使用 `mc`，那你可以直接略过本节，跳到[Step 4](#)。

添加一个或多个S3兼容的服务，请参考下面说明。`mc` 将所有的配置信息都存储在 `~/.mc/config.json` 文件中。

使用

```
mc config host add <ALIAS> <YOUR-S3-ENDPOINT> <YOUR-ACCESS-KEY> <YOUR-SECRET-KEY> [--api API-SIGNATURE]
```

别名就是给你的云存储服务起了一个短点的外号。S3 endpoint, access key和secret key是你的云存储服务提供的。API签名是可选参数， 默认情况下，它被设置为"S3v4"。

示例-MinIO云存储

从MinIO服务获得URL、access key和secret key。

```
mc config host add minio http://192.168.1.51 BKIKJAA5BMMU2RH06IBB V7f1CwQqAcwo80UEIJEjc5gVQUSSx5ohQ9GSrr12 --api s3v4
```

示例-AWS S3云存储

参考[AWS Credentials指南](#)获取你的AccessKeyId和SecretAccessKey。

```
mc config host add s3 https://s3.amazonaws.com BKIKJAA5BMMU2RH06IBB V7f1CwQqAcwo80UEIJEjc5gVQUSSx5ohQ9GSrr12 --api s3v4
```

示例-Google云存储

参考[Google Credentials Guide](#)获取你的AccessKeyId和SecretAccessKey。

```
mc config host add gcs https://storage.googleapis.com BKIKJAA5BMMU2RH06IBB V8f1CwQqAcwo80UEIJEjc5gVQUSSx5ohQ9GSrr12 --api s3v2
```

注意： Google云存储只支持旧版签名版本V2， 所以你需要选择S3v2。

4. 验证

mc 预先配置了云存储服务URL: <https://play.min.io>, 别名“play”。它是一个用于研发和测试的MinIO服务。如果想测试Amazon S3, 你可以将“play”替换为“s3”。

示例：

列出<https://play.min.io>上的所有存储桶。

```
mc ls play
[2016-03-22 19:47:48 PDT]    0B my-bucketname/
[2016-03-22 22:01:07 PDT]    0B mytestbucket/
[2016-03-22 20:04:39 PDT]    0B mybucketname/
[2016-01-28 17:23:11 PST]    0B newbucket/
[2016-03-20 09:08:36 PDT]    0B s3git-test/
```

5. 日常使用

你可以添加shell别名来覆盖默认的Unix工具命令。

```
alias ls='mc ls'
alias cp='mc cp'
alias cat='mc cat'
alias mkdir='mc mb'
alias pipe='mc pipe'
alias find='mc find'
```

6. 全局参数

参数 **[--debug]**

Debug参数开启控制台输出debug信息。

示例：输出 `ls` 命令的详细**debug**信息。

```
mc --debug ls play
mc: <DEBUG> GET / HTTP/1.1
Host: play.min.io
User-Agent: MinIO (darwin; amd64) minio-go/1.0.1 mc/2016-04-01T00:22:11Z
Authorization: AWS4-HMAC-SHA256 Credential=**REDACTED**/20160408/us-east-1/s3/aws4_request, SignedHeaders=exp
ect;host;x-amz-content-sha256;x-amz-date, Signature=**REDACTED**
Expect: 100-continue
X-Amz-Content-Sha256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
X-Amz-Date: 20160408T145236Z
Accept-Encoding: gzip

mc: <DEBUG> HTTP/1.1 200 OK
Transfer-Encoding: chunked
Accept-Ranges: bytes
Content-Type: text/xml; charset=utf-8
Date: Fri, 08 Apr 2016 14:54:55 GMT
Server: MinIO/DEVELOPMENT.2016-04-07T18-53-27Z (linux; amd64)
Vary: Origin
X-Amz-Request-Id: HP30I0W2U49BDBIO

mc: <DEBUG> Response Time: 1.220112837s

[...]

[2016-04-08 03:56:14 IST]      0B albums/
[2016-04-04 16:11:45 IST]      0B backup/
[2016-04-01 20:10:53 IST]      0B deebucket/
[2016-03-28 21:53:49 IST]      0B guestbucket/
```

参数 **[--json]**

JSON参数启用JSON格式的输出。

示例：列出MinIO `play`服务的所有存储桶。

```
mc --json ls play
{"status":"success","type":"folder","lastModified":"2016-04-08T03:56:14.577+05:30","size":0,"key":"albums/"}
{"status":"success","type":"folder","lastModified":"2016-04-04T16:11:45.349+05:30","size":0,"key":"backup/"}
{"status":"success","type":"folder","lastModified":"2016-04-01T20:10:53.941+05:30","size":0,"key":"deebucket/"}
{"status":"success","type":"folder","lastModified":"2016-03-28T21:53:49.217+05:30","size":0,"key":"guestbucke
t/"}
```

参数 **[--no-color]**

这个参数禁用颜色主题。对于一些比较老的终端有用。

参数 **[--quiet]**

这个参数关闭控制台日志输出。

参数 [**--config-dir**]

这个参数参数自定义的配置文件路径。

参数 [**--insecure**]

跳过SSL证书验证。

7. 命令

ls - 列出存储桶和对象	mb - 创建存储桶	cat - 合并对象
cp - 拷贝对象	rm - 删除对象	pipe - Pipe到一个对象
share - 共享	mirror - 存储桶镜像	find - 查找文件和对象
diff - 比较存储桶差异	policy - 给存储桶或前缀设置访问策略	
config - 管理配置文件	watch - 事件监听	events - 管理存储桶事件
update - 管理软件更新	version - 显示版本信息	

ls 命令 - 列出对象

ls 命令列出文件、对象和存储桶。使用 **--incomplete** flag 可列出未完整拷贝的内容。

用法:

```
mc ls [FLAGS] TARGET [TARGET ...]
```

FLAGS:

--help, -h	显示帮助。
--recursive, -r	递归。
--incomplete, -I	列出未完整上传的对象。

示例: 列出所有<https://play.min.io>上的存储桶。

```
mc ls play
[2016-04-08 03:56:14 IST]    0B albums/
[2016-04-04 16:11:45 IST]    0B backup/
[2016-04-01 20:10:53 IST]    0B deebucket/
[2016-03-28 21:53:49 IST]    0B guestbucket/
[2016-04-08 20:58:18 IST]    0B mybucket/
```

mb 命令 - 创建存储桶

mb 命令在对象存储上创建一个新的存储桶。在文件系统，它就和 **mkdir -p** 命令是一样的。存储桶相当于文件系统中的磁盘或挂载点，不应视为文件夹。MinIO对每个用户创建的存储桶数量没有限制。在Amazon S3上，每个帐户被限制为100个存储桶。有关更多信息，请参阅[S3上的存储桶限制和限制](#)。

用法:

```
mc mb [FLAGS] TARGET [TARGET...]
```

FLAGS:

--help, -h	显示帮助。
--region "us-east-1"	指定存储桶的region，默认是'us-east-1'。

示例：在<https://play.min.io>上创建一个名叫“mybucket”的存储桶。

```
mc mb play/mybucket
Bucket created successfully 'play/mybucket'.
```

cat 命令 - 合并对象

cat 命令将一个文件或者对象的内容合并到另一个上。你也可以用它将对象的内容输出到stdout。

用法：

```
mc cat [FLAGS] SOURCE [SOURCE...]
```

FLAGS:

```
--help, -h          显示帮助。
```

示例：显示 myobject.txt 文件的内容

```
mc cat play/mybucket/myobject.txt
Hello MinIO!!
```

pipe 命令 - Pipe到对象

pipe 命令拷贝stdin里的内容到目标输出，如果没有指定目标输出，则输出到stdout。

用法：

```
mc pipe [FLAGS] [TARGET]
```

FLAGS:

```
--help, -h          显示帮助。
```

示例：将MySQL数据库dump文件输出到Amazon S3。

```
mysqldump -u root -p ***** accountsdb | mc pipe s3/sql-backups/backups/accountsdb-oct-9-2015.sql
```

cp 命令 - 拷贝对象

cp 命令拷贝一个或多个源文件目标输出。所有到对象存储的拷贝操作都进行了MD4SUM checkSUM校验。可以从故障点恢复中断或失败的复制操作。

用法：

```
mc cp [FLAGS] SOURCE [SOURCE...] TARGET
```

FLAGS:

```
--help, -h          显示帮助。
--recursive, -r      递归拷贝。
```

示例：拷贝一个文本文件到对象存储。

```
mc cp myobject.txt play/mybucket
myobject.txt: 14 B / 14 B ██████████ 100.00 % 41 B/s 0
```

rm 命令 - 删除存储桶和对象。

使用 `rm` 命令删除文件对象或者存储桶。

用法:

```
mc rm [FLAGS] TARGET [TARGET ...]
```

FLAGS:

--help, -h	显示帮助。
--recursive, -r	递归删除。
--force	强制执行删除操作。
--prefix	删除批配这个前缀的对象。
--incomplete, -I	删除未完整上传的对象。
--fake	模拟一个假的删除操作。
--stdin	从STDIN中读对象列表。
--older-than value	删除N天前的对象（默认是0天）。

示例： 删除一个对象。

```
mc rm play/mybucket/myobject.txt
Removed 'play/mybucket/myobject.txt'.
```

示例： 删除一个存储桶并递归删除里面所有的内容。由于这个操作太危险了，你必须传 `--force` 参数指定强制删除。

```
mc rm --recursive --force play/myobject
Removed 'play/myobject/newfile.txt'.
Removed 'play/myobject/otherobject.txt'.
```

示例： 从 `mybucket` 里删除所有未完整上传的对象。

```
mc rm --incomplete --recursive --force play/mybucket
Removed 'play/mybucket/mydvd.iso'.
Removed 'play/mybucket/backup.tgz'.
```

示例： 删除一天前的对象。

```
mc rm --force --older-than=1 play/mybucket/oldsongs
```

share 命令 - 共享

`share` 命令安全地授予上传或下载的权限。此访问只是临时的，与远程用户和应用程序共享也是安全的。如果你想授予永久访问权限，你可以看看 `mc policy` 命令。

生成的网址中含有编码后的访问认证信息，任何企图篡改URL的行为都会使访问无效。想了解这种机制是如何工作的，请参考[Pre-Signed URL](#)技术。

用法:

```
mc share [FLAGS] COMMAND
```

FLAGS:

--help, -h	显示帮助。
------------	-------

COMMANDS:

download	生成有下载权限的URL。
upload	生成有上传权限的URL。
list	列出先前共享的对象和文件夹。

子命令 `share download` - 共享下载

`share download` 命令生成不需要access key和secret key即可下载的URL，过期参数设置成最大有效期（不大于7天），过期之后权限自动回收。

用法:

```
mc share download [FLAGS] TARGET [TARGET...]
```

FLAGS:

--help, -h	显示帮助。
--recursive, -r	递归共享所有对象。
--expire, -E "168h"	设置过期时限, NN[h m s]。

示例：生成一个对一个对象有4小时访问权限的URL。

```
mc share download --expire 4h play/mybucket/myobject.txt
URL: https://play.min.io/mybucket/myobject.txt
Expires: 0 days 4 hours 0 minutes 0 seconds
Share: https://play.min.io/myobject.txt?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=Q3AM3UQ867SPQQA43P2F%2F20160408%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20160408T182008Z&X-Amz-Expires=604800&X-Amz-SignedHeaders=host&X-Amz-Signature=1527fc8f21a3a7e39ce3c456907a10b389125047adc552bcd86630b9d459b634
```

子命令 `share upload` - 共享上传

`share upload` 命令生成不需要access key和secret key即可上传的URL。过期参数设置成最大有效期（不大于7天），过期之后权限自动回收。`Content-type`参数限制只允许上传指定类型的文件。

用法:

```
mc share upload [FLAGS] TARGET [TARGET...]
```

FLAGS:

--help, -h	显示帮助。
--recursive, -r	递归共享所有对象。
--expire, -E "168h"	设置过期时限, NN[h m s]。

示例：生成一个curl命令，赋予上传到`play/mybucket/myotherobject.txt`的权限。

```
mc share upload play/mybucket/myotherobject.txt
URL: https://play.min.io/mybucket/myotherobject.txt
Expires: 7 days 0 hours 0 minutes 0 seconds
Share: curl https://play.min.io/mybucket -F x-amz-date=20160408T182356Z -F x-amz-signature=de343934bd0ba38bda0903813b5738f23dde67b4065ea2ec2e4e52f6389e51e1 -F bucket=mybucket -F policy=eyJleHBpcmF0aW9uIjoiMjAxNjowNC0xNlVQxDoyMzo1NS4wMDdaIwiY29uZG10aw9ucyI6W1siZXElCIkYnVja2V0IiwibXlidWNrZXQiXSxbImVxIiwiJGtleSIsIm15b3RoZXJyVmplY3QudHh0I10sWyJlcSIsIiR4LWFtei1kYXR1IiwiMjAxNjA0MDhUMTgyMzU2WiJdLFsiZXElCIkeC1hbXotYwxb3JpdGhtIiwiQVdTNC1ITUFDLVNIQTlNiJdLFsiZXElCIkeC1hbXotY3J1ZGVudGlhbCIIsI1EzQU0zVVE4NjdTUFRQTQzUDJGLzIwMTYwNDA4L3VzLWVhc3QtMS9zMy9hd3M0X3J1cXV1c3QiXV19 -F x-amz-algorithm=AWS4-HMAC-SHA256 -F x-amz-credential=Q3AM3UQ867SPQQA43P2F/20160408/us-east-1/s3/aws4_request -F key=myotherobject.txt -F file=@<FILE>
```

子命令 `share list` - 列出之前的共享

`share list` 列出没过期的共享URL。

用法:

```
mc share list COMMAND
```

COMMAND:

```
upload: 列出先前共享的有上传权限的URL。  
download: 列出先前共享的有下载权限的URL。
```

mirror 命令 - 存储桶镜像

`mirror` 命令和 `rsync` 类似，只不过它是在文件系统和对象存储之间做同步。

用法:

```
mc mirror [FLAGS] SOURCE TARGET
```

FLAGS:

--help, -h	显示帮助。
--force	强制覆盖已经存在的目标。
--fake	模拟一个假的操作。
--watch, -w	监听改变并执行镜像操作。
--remove	删除目标上的外部的文件。

示例：将一个本地文件夹镜像到<https://play.min.io>上的'mybucket'存储桶。

```
mc mirror locadir/ play/mybucket  
locadir/b.txt: 40 B / 40 B |████████████████████████████████| 100.00 % 73 B/s 0
```

示例：持续监听本地文件夹修改并镜像到<https://play.min.io>上的'mybucket'存储桶。

```
mc mirror -w locadir play/mybucket  
locadir/new.txt: 10 MB / 10 MB |████████████████████████████████| 100.00 % 1 MB/s 15s
```

find 命令 - 查找文件和对象

`find` 命令通过指定参数查找文件，它只列出满足条件的数据。

用法:

```
mc find PATH [FLAGS]
```

FLAGS:

--help, -h	显示帮助。
--exec value	为每个匹配对象生成一个外部进程（请参阅FORMAT）
--name value	查找匹配通配符模式的对象。
...	
...	

示例：持续从s3存储桶中查找所有jpeg图像，并复制到minio "play/bucket"存储桶

```
mc find s3/bucket --name "*.jpg" --watch --exec "mc cp {} play/bucket"
```

diff 命令 - 显示差异

`diff` 命令计算两个目录之间的差异。它只列出缺少的或者大小不同的内容。

它不比较内容，所以可能的是，名称相同，大小相同但内容不同的对象没有被检测到。这样，它可以在不同站点或者大量数据的情况下快速比较。

用法:

```
mc diff [FLAGS] FIRST SECOND
```

```
FLAGS:  
--help, -h 显示帮助。
```

示例： 比较一个本地文件夹和一个远程对象存储服务

```
mc diff locaddir play/mybucket  
'locaddir/notes.txt' and 'https://play.min.io/mybucket/notes.txt' - only in first.
```

watch 命令 - 监听文件和对象存储事件。

watch 命令提供了一种方便监听对象存储和文件系统上不同类型事件的方式。

用法:

```
mc watch [FLAGS] PATH
```

FLAGS:

--events value	过滤不同类型的事件， 默认是所有类型的事件 (默认: "put,delete,get")
--prefix value	基于前缀过滤事件。
--suffix value	基于后缀过滤事件。
--recursive	递归方式监听事件。
--help, -h	显示帮助。

示例： 监听对象存储的所有事件

```
mc watch play/testbucket  
[2016-08-18T00:51:29.735Z] 2.7KiB ObjectCreated https://play.min.io/testbucket/CONTRIBUTING.md  
[2016-08-18T00:51:29.780Z] 1009B ObjectCreated https://play.min.io/testbucket/MAINTAINERS.md  
[2016-08-18T00:51:29.839Z] 6.9KiB ObjectCreated https://play.min.io/testbucket/README.md
```

示例： 监听本地文件夹的所有事件

```
mc watch ~/Photos  
[2016-08-17T17:54:19.565Z] 3.7MiB ObjectCreated /home/minio/Downloads/tmp/5467026530_a8611b53f9_o.jpg  
[2016-08-17T17:54:19.565Z] 3.7MiB ObjectCreated /home/minio/Downloads/tmp/5467026530_a8611b53f9_o.jpg  
...  
[2016-08-17T17:54:19.565Z] 7.5MiB ObjectCreated /home/minio/Downloads/tmp/8771468997_89b762d104_o.jpg
```

events 命令 - 管理存储桶事件通知。

events 提供了一种方便的配置存储桶的各种类型事件通知的方式。MinIO事件通知可以配置成使用 AMQP, Redis, ElasticSearch, NATS 和 PostgreSQL 服务。MinIO configuration 提供了如何配置的更多细节。

用法:

```
mc events COMMAND [COMMAND FLAGS | -h] [ARGUMENTS...]
```

COMMANDS:

add	添加一个新的存储桶通知。
remove	删除一个存储桶通知。使用'--force'可以删除所有存储桶通知。
list	列出存储桶通知。

FLAGS:

--help, -h	显示帮助。
------------	-------

示例： 列出所有存储桶通知。

```
mc events list play/andoria
MyTopic      arn:minio:sns:us-east-1:1:TestTopic    s3:ObjectCreated:*,s3:ObjectRemoved:*    suffix:.jpg
```

示例：添加一个新的'sqs'通知，仅接收*ObjectCreated*事件。

```
mc events add play/andoria arn:minio:sqs:us-east-1:1:your-queue --events put
```

示例：添加一个带有过滤器的'sqs'通知。

给 sqs 通知添加 prefix 和 suffix 过滤规则。

```
mc events add play/andoria arn:minio:sqs:us-east-1:1:your-queue --prefix photos/ --suffix .jpg
```

示例：删除一个'sqs'通知

```
mc events remove play/andoria arn:minio:sqs:us-east-1:1:your-queue
```

policy 命令 - 管理存储桶策略

管理匿名访问存储桶和其内部内容的策略。

用法:

```
mc policy [FLAGS] PERMISSION TARGET
mc policy [FLAGS] TARGET
mc policy list [FLAGS] TARGET
```

PERMISSION:

```
Allowed policies are: [none, download, upload, public].
```

FLAGS:

```
--help, -h          显示帮助。
```

示例：显示当前匿名存储桶策略

显示当前 `mybucket/myphotos/2020/` 子文件夹的匿名策略。

```
mc policy play/mybucket/myphotos/2020/
Access permission for 'play/mybucket/myphotos/2020/' is 'none'
```

示例：设置可下载的匿名存储桶策略。

设置 `mybucket/myphotos/2020/` 子文件夹可匿名下载的策略。现在，这个文件夹下的对象可被公开访问。比如：`mybucket/myphotos/2020/yourobjetcname` 可通过这个URL
<https://play.min.io/mybucket/myphotos/2020/yourobjetcname>访问。

```
mc policy set download play/mybucket/myphotos/2020/
Access permission for 'play/mybucket/myphotos/2020/' is set to 'download'
```

示例：删除当前的匿名存储桶策略

删除所有 `mybucket/myphotos/2020/` 这个子文件夹下的匿名存储桶策略。

```
mc policy set none play/mybucket/myphotos/2020/
Access permission for 'play/mybucket/myphotos/2020/' is set to 'none'
```

config 命令 - 管理配置文件

`config host` 命令提供了一个方便地管理 `~/.mc/config.json` 配置文件中的主机信息的方式，你也可以用文本编辑器手动修改这个配置文件。

用法:

```
mc config host COMMAND [COMMAND FLAGS | -h] [ARGUMENTS...]
```

COMMANDS:

- `add, a` 添加一个新的主机到配置文件。
- `remove, rm` 从配置文件中删除一个主机。
- `list, ls` 列出配置文件中的主机。

FLAGS:

- `--help, -h` 显示帮助。

示例: 管理配置文件

添加MinIO服务的access和secret key到配置文件，注意，`shell`的`history`特性可能会记录这些信息，从而带来安全隐患。在`bash shell`，使用`set -o` 和 `set +o` 来关闭和开启`history`特性。

```
set +o history
mc config host add myminio http://localhost:9000 OMQAGGOL63D7UNVQFY8X GcY5RHNmnElwvD/1QxD3spEIGj+Vt9L7eHaAaBTk
J
set -o history
```

update 命令 - 软件更新

从<https://dl.min.io>检查软件更新。`Experimental`标志会检查`unstable`实验性的版本，通常用作测试用途。

用法:

```
mc update [FLAGS]
```

FLAGS:

- `--quiet, -q` 关闭控制台输出。
- `--json` 使用JSON格式输出。
- `--help, -h` 显示帮助。

示例: 检查更新

```
mc update
You are already running the most recent version of 'mc'.
```

version 命令 - 显示版本信息

显示当前安装的 `mc` 版本。

用法:

```
mc version [FLAGS]
```

FLAGS:

- `--quiet, -q` 关闭控制台输出。
- `--json` 使用JSON格式输出。
- `--help, -h` 显示帮助。

示例: 输出`mc`版本。

```
mc version
Version: 2016-04-01T00:22:11Z
Release-tag: RELEASE.2016-04-01T00-22-11Z
Commit-id: 12adf3be326f5b6610cdd1438f72dfd861597fce
```

MinIO 管理员完整指南

MinIO Client (mc) 提供了“admin”子命令来对您的MinIO部署执行管理任务。

service	服务重启并停止所有MinIO服务器
update	更新所有MinIO服务器
info	信息显示MinIO服务器信息
user	用户管理用户
group	小组管理小组
policy	MinIO服务器中定义的策略管理策略
config	配置管理MinIO服务器配置
heal	修复MinIO服务器上的磁盘，存储桶和对象
profile	概要文件生成概要文件数据以进行调试
top	顶部提供MinIO的顶部统计信息
trace	跟踪显示MinIO服务器的http跟踪
console	控制台显示MinIO服务器的控制台日志
prometheus	Prometheus管理Prometheus配置
kms	kms执行KMS管理操作

1. 下载MinIO Client

Docker稳定版

```
docker pull minio / mc
docker run minio / mc admin info play
```

Docker Edge

```
docker pull minio / mc: edge
docker run minio / mc:edge admin info server play
```

Homebrew (macOS)

使用[Homebrew] (<http://brew.sh/>) 安装mc软件包

```
brew install minio/stable/mc
mc --help
```

二进制下载 (GNU / Linux)

平台	CPU架构	URL
GNU/Linux	64-bit Intel	http://dl.minio.org.cn/client/mc/release/linux-amd64/mc
64-bit PPC		http://dl.minio.org.cn/client/mc/release/linux-ppc64le/mc

```
chmod +x mc
./mc --help
```

二进制下载 (Microsoft Windows)

平台	CPU架构	URL
Microsoft Windows	64-bit Intel	http://dl.minio.org.cn/client/mc/release/windows-amd64/mc.exe

```
mc.exe --help
```

从源代码安装

源代码安装仅适用于开发人员和高级用户。 **mc update**命令不支持基于源的安装的更新通知。请从 <https://min.io/download/#minio-client> 下载官方版本。

如果您没有可用的Golang环境, 请遵循[如何安装Golang] (<https://golang.org/doc/install>) 。

```
go get -d github.com/minio/mc  
cd ${GOPATH}/src/github.com/minio/mc  
make
```

2.运行MinIO Client

GNU / Linux

```
chmod +x mc  
./mc --help
```

macOS

```
chmod 755 mc  
./mc --help
```

Microsoft Windows

```
mc.exe --help
```

3.添加MinIO存储服务

MinIO服务器显示URL, 访问权和秘密密钥。

用法

```
mc config host add <ALIAS> <YOUR-MINIO-ENDPOINT> [YOUR-ACCESS-KEY] [YOUR-SECRET-KEY]
```

键必须由参数或标准输入提供。

别名只是您的MinIO服务的简称。 MinIO端点, 访问和密钥由您的MinIO服务提供。 Admin API使用“S3v4”签名, 无法更改。

例子

1.参数键

```
mc config host add minio http://192.168.1.51:9000 BKIKJAA5BMMU2RHO6IBB V7f1CwQqAcwo80UEIJEjc5gVQUSSx5ohQ9GSrr
```

2.按键提示

```
mc config host add minio http://192.168.1.51:9000
Enter Access Key: BKIKJAA5BMMU2RH06IBB
Enter Secret Key: V7f1CwQqAcwo80UEIJEjc5gVQUSSx5ohQ9GSrr12
```

2.管道按键

```
echo -e "BKIKJAA5BMMU2RH06IBB\nV7f1CwQqAcwo80UEIJEjc5gVQUSSx5ohQ9GSrr12" | \
mc config host add minio http://192.168.1.51:9000
```

4.测试您的设置

例：

获取已配置别名“minio”的MinIO服务器信息

```
mc admin info minio
• min.minio.io
  Uptime: 11 hours
  Version: 2020-01-17T22:08:02Z
  Network: 1/1 OK
  Drives: 4/4 OK

  2.1 GiB Used, 158 Buckets, 12,092 Objects
  4 drives online, 0 drives offline
```

5.日常使用

您可以添加外壳别名以获取信息，以便恢复。

```
alias minfo='mc admin info'
alias mheal='mc admin heal'
```

6.全局选项

选项[**--debug**]

调试选项使调试输出到控制台。

示例：显示“*info*”命令的详细调试输出。

```
mc: <DEBUG> GET /minio/admin/v2/info HTTP/1.1
Host: play.minio.io
User-Agent: MinIO (linux; amd64) madmin-go/0.0.1 mc/DEVELOPMENT.GOGET
Authorization: AWS4-HMAC-SHA256 Credential=**REDACTED**/20200120//s3/aws4_request, SignedHeaders=host;x-amz-content-sha256;x-amz-date, Signature=**REDACTED**
X-Amz-Content-Sha256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
X-Amz-Date: 20200120T185844Z
Accept-Encoding: gzip

mc: <DEBUG> HTTP/1.1 200 OK
Content-Length: 1105
```

```
Accept-Ranges: bytes
Connection: keep-alive
Content-Security-Policy: block-all-mixed-content
Content-Type: application/json
Date: Mon, 20 Jan 2020 18:58:44 GMT
Server: nginx/1.10.3 (Ubuntu)
Vary: Origin
X-Amz-Bucket-Region: us-east-1
X-Amz-Request-Id: 15EBAD6087210B2A
X-Xss-Protection: 1; mode=block
```

```
mc: <DEBUG> Response Time: 381.860854ms
```

- play.minio.io
Uptime: 11 hours
Version: 2020-01-17T22:08:02Z
Network: 1/1 OK
Drives: 4/4 OK

```
2.1 GiB Used, 158 Buckets, 12,092 Objects
4 drives online, 0 drives offline
```

选项[**--json**]

JSON选项启用[JSON行] (<http://jsonlines.org/>) 格式的可解析输出。

*示例：MinIO服务器

```
mc admin --json info play
{
    "status": "success",
    "info": {
        "mode": "online",
        "region": "us-east-1",
        "deploymentID": "728e91fd-ed0c-4500-b13d-d143561518bf",
        "buckets": {
            "count": 158
        },
        "objects": {
            "count": 12092
        },
        "usage": {
            "size": 2249526349
        },
        "services": {
            "vault": {
                "status": "KMS configured using master key"
            },
            "ldap": {}
        },
        "backend": {
            "backendType": "Erasure",
            "onlineDisks": 4,
            "rrSCData": 2,
            "rrSCPParity": 2,
            "standardSCData": 2,
            "standardSCPParity": 2
        },
        "servers": [
            {

```

```
        "state": "ok",
        "endpoint": "play.minio.io",
        "uptime": 41216,
        "version": "2020-01-17T22:08:02Z",
        "commitID": "b0b25d558e25608e3a604888a0a43e58e8301dfb",
        "network": {
            "play.minio.io": "online"
        },
        "disks": [
            {
                "path": "/home/play/data1",
                "state": "ok",
                "uuid": "c1f8dbf8-39c8-46cd-bab6-2c87d18db06a",
                "totalspace": 8378122240,
                "usedspace": 1410588672
            },
            {
                "path": "/home/play/data2",
                "state": "ok",
                "uuid": "9616d28f-5f4d-47f4-9c6d-4deb0da07cad",
                "totalspace": 8378122240,
                "usedspace": 1410588672
            },
            {
                "path": "/home/play/data3",
                "state": "ok",
                "uuid": "4c822d68-4d9a-4fa3-aabb-5bf5a58e5848",
                "totalspace": 8378122240,
                "usedspace": 1410588672
            },
            {
                "path": "/home/play/data4",
                "state": "ok",
                "uuid": "95b5a33c-193b-4a11-b13a-a99bc1483182",
                "totalspace": 8378122240,
                "usedspace": 1410588672
            }
        ]
    }
}
```

选项[**--no-color**]

此选项禁用颜色主题。对于哑终端很有用。

选项[**--quiet**]

安静选项禁止显示聊天控制台输出。

选项[**--config-dir**]

使用此选项设置自定义配置路径。

选项[**--insecure**]

跳过SSL证书验证。

7. 命令

命令
service - 重新启动和停止所有MinIO服务器
update - 更新所有MinIO服务器
info - 显示MinIO服务器信息
user - 管理用户
group - 管理组
policy - 管理固定政策
config - 管理服务器配置文件
heal - 修复MinIO服务器上的磁盘，存储桶和对象
profile - 生成用于调试目的的配置文件数据
top - 为MinIO提供类似顶部的统计信息
trace - 显示MinIO服务器的http跟踪
console - 显示MinIO服务器的控制台日志
prometheus - 管理prometheus配置设置

命令 **update** -更新所有**MinIO**服务器

`update`命令提供了一种更新集群中所有MinIO服务器的方法。您还可以使用带有 `update` 命令的私有镜像服务器来更新MinIO集群。如果MinIO在无法访问Internet的环境中运行，这很有用。

示例：更新所有**MinIO**服务器。

```
mc admin update play
Server `play` updated successfully from RELEASE.2019-08-14T20-49-49Z to RELEASE.2019-08-21T19-59-10Z
```

使用私有镜像更新**MinIO**的步骤

为了在私有镜像服务器上使用 `update` 命令，您需要在私有镜像服务器上的<https://dl.minio.io/server/minio/release/linux-amd64/>上镜像目录结构，然后提供：

```
mc admin update myminio https://myfavorite-mirror.com/minio-server/linux-amd64/minio.sha256sum
Server `myminio` updated successfully from RELEASE.2019-08-14T20-49-49Z to RELEASE.2019-08-21T19-59-10Z
```

注意： -指向分布式安装程序的别名，此命令将自动更新群集中的所有MinIO服务器。 - `update` 是您的MinIO服务的破坏性操作，任何正在进行的API操作都将被强制取消。因此，仅在计划为部署进行MinIO升级时才应使用它。 -建议在更新成功完成后执行重新启动。

命令 **service** -重新启动并停止所有**MinIO**服务器

服务命令提供了一种重新启动和停止所有MinIO服务器的方法。

注意： -指向分布式设置的别名，此命令将在所有服务器上自动执行相同的操作。 - `restart` 和 `stop` 子命令是MinIO服务的破坏性操作，任何正在进行的API操作都将被强制取消。因此，仅应在管理环境下使用。请谨慎使用。

```
NAME:
  mc admin service - restart and stop all MinIO servers
```

```
FLAGS:
```

```
--help, -h           show help

COMMANDS:
restart  restart all MinIO servers
stop     stop all MinIO servers
```

示例：重新启动所有*MinIO*服务器。

```
mc admin service restart play
Restarted `play` successfully.
```

命令 **info** -显示*MinIO*服务器信息

“**info**”命令显示一台或多台*MinIO*服务器的服务器信息（在分布式集群下）

```
NAME:
mc admin info - get MinIO server information

FLAGS:
--help, -h           show help
```

示例：显示*MinIO*服务器信息。

```
mc admin info play
• play.minio.io
Uptime: 11 hours
Version: 2020-01-17T22:08:02Z
Network: 1/1 OK
Drives: 4/4 OK

2.1 GiB Used, 158 Buckets, 12,092 Objects
4 drives online, 0 drives offline
```

命令 **policy** -管理固定策略

policy命令，用于添加，删除，列出策略，获取有关策略的信息并为*MinIO*服务器上的用户设置策略。

```
NAME:
mc admin policy - manage policies

FLAGS:
--help, -h           show help

COMMANDS:
add    add new policy
remove remove policy
list   list all policies
info   show info on a policy
set    set IAM policy on a user or group
```

示例：列出*MinIO*上的所有固定策略。

```
mc admin policy list myminio/
diagnostics
readonly
readwrite
```

writeonly

示例：在MinIO上添加新策略'listbucketsonly'，策略来自/tmp/listbucketsonly.json。 *对用户应用此策略时，该用户只能列出顶层存储桶，而不能列出其他内容，没有前缀，没有对象。

首先使用以下信息创建json文件/tmp/listbucketsonly.json。

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "s3>ListAllMyBuckets"  
            ],  
            "Resource": [  
                "arn:aws:s3:::*"  
            ]  
        }  
    ]  
}
```

将策略作为“listbucketsonly”添加到策略数据库中

```
mc admin policy add myminio/ listbucketsonly /tmp/listbucketsonly.json  
Added policy `listbucketsonly` successfully.
```

示例：在MinIO上删除策略“listbucketsonly”。

```
mc admin policy remove myminio/ listbucketsonly  
Removed policy `listbucketsonly` successfully.
```

示例：显示罐头策略的信息，“只写”

```
mc admin policy info myminio/ writeonly  
{"Version":"2012-10-17","Statement":[{"Effect":"Allow","Action":["s3:PutObject"],"Resource":["arn:aws:s3:::*"]}]}
```

示例：在用户或组上设置固定策略。“只写”

```
mc admin policy set myminio/ writeonly user=someuser  
Policy writeonly is set on user `someuser`  
  
mc admin policy set myminio/ writeonly group=somegroup  
Policy writeonly is set on group `somegroup`
```

命令 **user** -管理用户

用户命令，用于添加，删除，启用，禁用MinIO服务器上的用户。

```
NAME:  
  mc admin user - manage users  
  
FLAGS:  
  --help, -h          show help
```

```
COMMANDS:  
add      add new user  
disable  disable user  
enable   enable user  
remove   remove user  
list     list all users  
info     display info of a user
```

示例：在MinIO上添加新用户'newuser'。

```
mc admin user add myminio/ newuser newuser123
```

示例：使用标准输入在MinIO上添加新用户'newuser'。

```
mc admin user add myminio/  
Enter Access Key: newuser  
Enter Secret Key: newuser123
```

示例：在MinIO上禁用用户“newuser”。

```
mc admin user disable myminio/ newuser
```

示例：在MinIO上启用用户“newuser”。

```
mc admin user enable myminio/ newuser
```

示例：在MinIO上删除用户'newuser'。

```
mc admin user remove myminio/ newuser
```

示例：列出MinIO上的所有用户。

```
mc admin user list --json myminio/  
{"status":"success","accessKey":"newuser","userStatus":"enabled"}
```

示例：显示用户信息

```
mc admin user info myminio someuser
```

命令 **group** -管理组

使用group命令在MinIO服务器上添加，删除，信息，列出，启用，禁用组。

```
NAME:  
  mc admin group - manage groups  
  
USAGE:  
  mc admin group COMMAND [COMMAND FLAGS | -h] [ARGUMENTS...]  

```

```
enable  Enable a group  
disable Disable a group
```

示例：将一对用户添加到*MinIO*上的“*somegroup*”组中。如果组不存在，则会创建该组。

```
mc admin group add myminio somegroup someuser1 someuser2
```

示例：从*MinIO*的“*somegroup*”组中删除一对用户。

```
mc admin group remove myminio somegroup someuser1 someuser2
```

示例：在*MinIO*上删除组“*somegroup*”。仅在给定组为空时有效。

```
mc admin group remove myminio somegroup
```

示例：在*MinIO*上获取有关“*somegroup*”组的信息。

```
mc admin group info myminio somegroup
```

示例：列出*MinIO*上的所有组。

```
mc admin group list myminio
```

示例：在*MinIO*上启用组“*somegroup*”。

```
mc admin group enable myminio somegroup
```

示例：在*MinIO*上禁用组“*somegroup*”。

```
mc admin group disable myminio somegroup
```

命令 **config** - 管理服务器配置

config命令用于管理*MinIO*服务器配置。

```
NAME:  
  mc admin config - manage configuration file  
  
USAGE:  
  mc admin config COMMAND [COMMAND FLAGS | -h] [ARGUMENTS...]  
  
COMMANDS:  
  get      get config of a MinIO server/cluster.  
  set      set new config file to a MinIO server/cluster.  
  
FLAGS:  
  --help, -h          Show help.
```

示例：获取*MinIO*服务器/集群的服务器配置。

```
mc admin config get myminio > /tmp/my-serverconfig
```

示例：设置*MinIO*服务器/集群的服务器配置。

```
mc admin config set myminio < /tmp/my-serverconfig
```

命令 `heal` -修复MinIO服务器上的磁盘，存储桶和对象

使用`heal`命令修复MinIO服务器上的磁盘，丢失的存储桶和对象。注意：此命令仅适用于MinIO擦除编码设置（独立和分布式）。

服务器已经有一个浅色的后台进程，可以在必要时修复磁盘，存储桶和对象。但是，它不会检测某些类型的数据损坏，尤其是很少发生的数据损坏，例如静默数据损坏。在这种情况下，您需要隔一段时间手动运行提供以下标志的`heal`命令：`--scan deep`。

要显示后台恢复过程的状态，只需键入以下命令：`mc admin heal your-alias`。

要扫描和修复所有内容，请输入：`mc admin heal -r your-alias`。

```
NAME:  
  mc admin heal - heal disks, buckets and objects on MinIO server  
  
FLAGS:  
  --scan value          select the healing scan mode (normal/deep) (default: "normal")  
  --recursive, -r        heal recursively  
  --dry-run, -n          only inspect data, but do not mutate  
  --force-start, -f      force start a new heal sequence  
  --force-stop, -s       force stop a running heal sequence  
  --remove               remove dangling objects in heal sequence  
  --help, -h              show help
```

示例：更换新磁盘后修复MinIO集群，递归修复所有存储桶和对象，其中“`myminio`”是MinIO服务器别名。

```
mc admin heal -r myminio
```

示例：递归修复特定存储桶上的MinIO集群，其中“`myminio`”是MinIO服务器别名。

```
mc admin heal -r myminio/mybucket
```

示例：递归修复特定对象前缀上的MinIO集群，其中“`myminio`”是MinIO服务器别名。

```
mc admin heal -r myminio/mybucket/myobjectprefix
```

示例：显示MinIO集群中自我修复过程的状态。

```
mc admin heal myminio/
```

命令 `profile` -生成用于调试目的的配置文件数据

```
NAME:  
  mc admin profile - generate profile data for debugging purposes  
  
COMMANDS:  
  start  start recording profile data  
  stop   stop and download profile data
```

开始进行CPU分析

```
mc admin profile start --type cpu myminio/
```

命令 **top** -为MinIO提供类似**top**的统计信息

注意：此命令仅适用于分布式MinIO设置。单节点和网关部署不支持此功能。

```
NAME:  
  mc admin top - provide top like statistics for MinIO  
  
COMMANDS:  
  locks  Get a list of the 10 oldest locks on a MinIO cluster.
```

示例：获取分布式*MinIO*群集上10个最旧锁的列表，其中‘*myminio*’是*MinIO*群集别名。

```
mc admin top locks myminio
```

命令 **trace** -显示MinIO服务器的http跟踪

trace命令显示一台或所有MinIO服务器（在分布式集群下）的服务器http跟踪

```
NAME:  
  mc admin trace - show http trace for MinIO server  
  
FLAGS:  
  --verbose, -v          print verbose trace  
  --all, -a              trace all traffic (including internode traffic between MinIO servers)  
  --errors, -e            trace failed requests only  
  --help, -h              show help
```

示例：显示*MinIO*服务器http跟踪。

```
mc admin trace myminio  
172.16.238.1 [REQUEST (objectAPIHandlers).ListBucketsHandler-fm] [154828542.525557] [2019-01-23 23:17:05 +0000]  
0]  
172.16.238.1 GET /  
172.16.238.1 Host: 172.16.238.3:9000  
172.16.238.1 X-Amz-Date: 20190123T231705Z  
172.16.238.1 Authorization: AWS4-HMAC-SHA256 Credential=minio/20190123/us-east-1/s3/aws4_request, SignedHeaders=host;x-amz-content-sha256;x-amz-date, Signature=8385097f264efaf1b71a9b56514b8166bb0a03af8552f83e2658f877776c46b3  
172.16.238.1 User-Agent: MinIO (linux; amd64) minio-go/v6.0.8 mc/2019-01-23T23:15:38Z  
172.16.238.1 X-Amz-Content-Sha256: e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855  
172.16.238.1  
172.16.238.1 <BODY>  
172.16.238.1 [RESPONSE] [154828542.525557] [2019-01-23 23:17:05 +0000]  
172.16.238.1 200 OK  
172.16.238.1 X-Amz-Request-Id: 157C9D641F42E547  
172.16.238.1 X-Minio-Deployment-Id: 5f20fd91-6880-455f-a26d-07804b6821ca  
172.16.238.1 X-Xss-Protection: 1; mode=block  
172.16.238.1 Accept-Ranges: bytes  
172.16.238.1 Content-Security-Policy: block-all-mixed-content  
172.16.238.1 Content-Type: application/xml  
172.16.238.1 Server: MinIO/RELEASE.2019-09-05T23-24-38Z  
172.16.238.1 Vary: Origin  
...
```

命令 **console** -显示**MinIO**服务器的控制台日志

“**console**”命令显示一台或所有**MinIO**服务器的服务器日志（在分布式集群下）

```
NAME:  
  mc admin console - show console logs for MinIO server  
  
FLAGS:  
  --limit value, -l value      show last n log entries (default: 10)  
  --help, -h                  show help
```

示例：显示**MinIO**服务器*http*跟踪。

```
mc admin console myminio  
  
API: SYSTEM(bucket=images)  
Time: 22:48:06 PDT 09/05/2019  
DeploymentID: 6faedeb5-5cf3-4133-8a37-07c5d500207c  
RequestID: <none>  
RemoteHost: <none>  
UserAgent: <none>  
Error: ARN 'arn:minio:sqs:us-east-1:1:webhook' not found  
        4: cmd/notification.go:1189:cmd.readNotificationConfig()  
        3: cmd/notification.go:780:cmd.(*NotificationSys).refresh()  
        2: cmd/notification.go:815:cmd.(*NotificationSys).Init()  
        1: cmd/server-main.go:375:cmd.serverMain()
```

命令 **prometheus** -管理**prometheus**配置设置

“**generate**”命令生成**prometheus**配置（要粘贴到“**prometheus.yml**”中）

```
NAME:  
  mc admin prometheus - manages prometheus config  
  
USAGE:  
  mc admin prometheus COMMAND [COMMAND FLAGS | -h] [ARGUMENTS...]  
  
COMMANDS:  
  generate generates prometheus config
```

示例：为生成**prometheus**配置。

```
mc admin prometheus generate <alias>  
- job_name: minio-job  
  bearer_token: <token>  
  metrics_path: /minio/prometheus/metrics  
  scheme: http  
  static_configs:  
    - targets: ['localhost:9000']
```

命令 **kms** -执行**KMS**管理操作

kms命令可用于执行**KMS**管理操作。

```
NAME:  
  mc admin kms - perform KMS management operations
```

```
USAGE:  
mc admin kms COMMAND [COMMAND FLAGS | -h] [ARGUMENTS...]
```

key子命令可用于执行主密钥管理操作。

```
NAME:  
mc admin kms key - manage KMS keys  
  
USAGE:  
mc admin kms key COMMAND [COMMAND FLAGS | -h] [ARGUMENTS...]
```

示例：显示默认主键的状态信息

```
mc admin kms key status play  
Key: my-minio-key  
• Encryption ✓  
• Decryption ✓
```

示例：显示一个特定主键的状态信息

```
mc admin kms key status play test-key-1  
Key: test-key-1  
• Encryption ✓  
• Decryption ✓
```

适用于Amazon S3兼容云存储的Minio JavaScript Library

MinIO JavaScript Client SDK提供简单的API来访问任何Amazon S3兼容的对象存储服务。

本快速入门指南将向您展示如何安装客户端SDK并执行示例JavaScript程序。有关API和示例的完整列表，请参阅[JavaScript客户端API参考文档](#)。

本文假设你已经安装了[nodejs](#)。

使用NPM下载

```
npm install --save minio
```

下载并安装源码

```
git clone https://github.com/minio/minio-js
cd minio-js
npm install
npm install -g
```

初始化Minio Client

你需要设置5个属性来链接Minio对象存储服务。

参数	描述
endPoint	对象存储服务的URL
port	TCP/IP端口号。可选值，如果是使用HTTP的话，默认值是 80；如果使用HTTPS的话，默认值是 443。
accessKey	Access key是唯一标识你的账户的用户ID。
secretKey	Secret key是你账户的密码。
useSSL	true代表使用HTTPS

```
var Minio = require('minio')

var minioClient = new Minio.Client({
    endPoint: 'play.min.io',
    port: 9000,
    useSSL: true,
    accessKey: 'Q3AM3UQ867SPQQA43P2F',
    secretKey: 'zuf+tfeSlswRu7BJ86wekitnifILbZam1KYY3TG'
});
```

示例-文件上传

本示例连接到一个对象存储服务，创建一个存储桶并上传一个文件到存储桶中。

我们在本示例中使用运行在<https://play.min.io>上的Minio服务，你可以用这个服务来开发和测试。示例中的访问凭据是公开的。

file-uploader.js

```

var Minio = require('minio')

// Instantiate the minio client with the endpoint
// and access keys as shown below.
var minioClient = new Minio.Client({
    endPoint: 'play.min.io',
    port: 9000,
    useSSL: true,
    accessKey: 'Q3AM3UQ867SPQQA43P2F',
    secretKey: 'zuf+tfeS1swRu7BJ86wekitnifILbZam1KYY3TG'
});

// File that needs to be uploaded.
var file = '/tmp/photos-europe.tar'

// Make a bucket called europetrip.
minioClient.makeBucket('europetrip', 'us-east-1', function(err) {
    if (err) return console.log(err)

    console.log('Bucket created successfully in "us-east-1".')

    var metaData = {
        'Content-Type': 'application/octet-stream',
        'X-Amz-Meta-Testing': 1234,
        'example': 5678
    }
    // Using fPutObject API upload your file to the bucket europetrip.
    minioClient.fPutObject('europetrip', 'photos-europe.tar', file, metaData, function(err, etag) {
        if (err) return console.log(err)
        console.log('File uploaded successfully.')
    });
});

```

运行file-uploader

```

node file-uploader.js
Bucket created successfully in "us-east-1".

mc ls play/europetrip/
[2016-05-25 23:49:50 PDT]  17MiB photos-europe.tar

```

API文档

完整的API文档在这里。

- [完整API文档](#)

API文档 : 操作存储桶

- [makeBucket](#)
- [listBuckets](#)
- [bucketExists](#)
- [removeBucket](#)
- [listObjects](#)
- [listObjectsV2](#)
- [listIncompleteUploads](#)

API 文档：操作文件对象

- `fPutObject`
- `fGetObject`

API 文档：操作对象

- `getObject`
- `putObject`
- `Object`
- `statObject`
- `removeObject`
- `removeIncompleteUpload`

API 文档：Presigned 操作

- `presignedGetObject`
- `presignedPutObject`
- `presignedPostPolicy`

API 文档：存储桶通知

- `getBucketNotification`
- `setBucketNotification`
- `removeAllBucketNotification`
- `listenBucketNotification` (MinIO Extension)

API 文档：存储桶策略

- `getBucketPolicy`
- `setBucketPolicy`

完整示例

完整示例：操作存储桶

- `list-buckets.js`
- `list-objects.js`
- `list-objects-v2.js`
- `bucket-exists.js`
- `make-bucket.js`
- `remove-bucket.js`
- `list-incomplete-uploads.js`

完整示例：操作文件对象

- `fput-object.js`
- `fget-object.js`

完整示例：操作对象

- `put-object.js`
- `get-object.js`
- `-object.js`

- [get-partialobject.js](#)
- [remove-object.js](#)
- [remove-incomplete-upload.js](#)
- [stat-object.js](#)

完整示例：**Presigned**操作

- [presigned-getobject.js](#)
- [presigned-putobject.js](#)
- [presigned-postpolicy.js](#)

完整示例：存储桶通知

- [get-bucket-notification.js](#)
- [set-bucket-notification.js](#)
- [remove-all-bucket-notification.js](#)
- [listen-bucket-notification.js](#) (MinIO Extension)

完整示例：存储桶策略

- [get-bucket-policy.js](#)
- [set-bucket-policy.js](#)

了解更多

- 完整文档
- [MinIO JavaScript Client SDK API文档](#)
- [创建属于你的购物APP-完整示例](#)

贡献

[贡献者指南](#)

JavaScript Client API参考文档

初始化Minio Client object.

MinIO

```
var Minio = require('minio')

var minioClient = new Minio.Client({
  endPoint: 'play.min.io',
  port: 9000,
  useSSL: true,
  accessKey: 'Q3AM3UQ867SPQQA43P2F',
  secretKey: 'zuf+tfteSlsRu7BJ86wekitnifILbZam1KYY3TG'
});
```

AWS S3

```
var Minio = require('minio')

var s3Client = new Minio.Client({
  endPoint: 's3.amazonaws.com',
  accessKey: 'YOUR-ACCESSKEYID',
  secretKey: 'YOUR-SECRETACCESSKEY'
})
```

操作存储桶	操作对象	Presigned操作	存储桶策略/通知
makeBucket	getObject	presignedUrl	getBucketNotification
listBuckets	getPartialObject	presignedGetObject	setBucketNotification
bucketExists	fGetObject	presignedPutObject	removeAllBucketNotification
removeBucket	putObject	presignedPostPolicy	getBucketPolicy
listObjects	fPutObject		setBucketPolicy
listObjectsV2	Object		listenBucketNotification
listIncompleteUploads	statObject		
	removeObject		
	removeIncompleteUpload		

1. 构造函数

`new Minio.Client ({endPoint, port, useSSL, accessKey, secretKey})`

```
new Minio.Client ({endPoint, port, useSSL, accessKey, secretKey})
```

初始化一个新的client对象。

参数

参数	类型	描述

<code>endPoint</code>	<code>string</code>	<code>endPoint</code> 是一个主机名或者IP地址。
<code>port</code>	<code>number</code>	TCP/IP端口号。可选，默认值是，如果是 <code>http</code> ,则默认80端口，如果是 <code>https</code> ,则默认是443端口。
<code>accessKey</code>	<code>string</code>	<code>accessKey</code> 类似于用户ID，用于唯一标识你的账户。
<code>secretKey</code>	<code>string</code>	<code>secretKey</code> 是你账户的密码。
<code>useSSL</code>	<code>bool</code>	如果是 <code>true</code> ，则用的是 <code>https</code> 而不是 <code>http</code> ,默认值是 <code>true</code> 。
<code>region</code>	<code>string</code>	设置该值以覆盖自动发现存储桶 <code>region</code> 。（可选）
<code>transport</code>	<code>string</code>	Set this value to pass in a custom transport. (Optional) - To be translated
<code>sessionToken</code>	<code>string</code>	Set this value to provide x-amz-security-token (AWS S3 specific). (Optional) - To be translated
<code>partSize</code>	<code>number</code>	Set this value to override default part size of 64MB for multipart uploads. (Optional) - To be translated

示例

创建连接Minio Server的客户端

```
var Minio = require('minio')

var minioClient = new Minio.Client({
    endPoint: 'play.min.io',
    port: 9000,
    useSSL: true,
    accessKey: 'Q3AM3UQ867SPQQA43P2F',
    secretKey: 'zuf+tfteS1swRu7BJ86wekitnifILbZam1KYY3TG'
});
```

创建连接AWS S3的客户端

```
var Minio = require('minio')

var s3Client = new Minio.Client({
    endPoint: 's3.amazonaws.com',
    accessKey: 'YOUR-ACCESSKEYID',
    secretKey: 'YOUR-SECRETACCESSKEY'
})
```

2. 操作存储桶

makeBucket(bucketName, region[, callback])

创建一个新的存储桶。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>region</code>	<code>string</code>	存储桶被创建的region(地区)，默认是us-east-1(美国东一区)，下面列举的是其它合法的值：
		<code>us-east-1</code>
		<code>us-west-1</code>

		us-west-2
		eu-west-1
		eu-central-1
		ap-southeast-1
		ap-northeast-1
		ap-southeast-2
		sa-east-1
		cn-north-1
callback(err)	function	回调函数， <code>err</code> 做为错误信息参数。如果创建存储桶成功则 <code>err</code> 为 <code>null</code> 。如果没有传 <code>callback</code> 的话，则返回一个 <code>Promise</code> 对象。

示例

```
minioClient.makeBucket('mybucket', 'us-east-1', function(err) {
  if (err) return console.log('Error creating bucket.', err)
  console.log('Bucket created successfully in "us-east-1".')
})
```

listBuckets([callback])

列出所有存储桶。

参数

参数	类型	描述
callback(<code>err</code> , <code>bucketStream</code>)	function	回调函数，第一个参数是错误信息。 <code>bucketStream</code> 是带有存储桶信息的流。如果没有传 <code>callback</code> 的话，则返回一个 <code>Promise</code> 对象。

`bucketStream` 格式如下:-

参数	类型	描述
<code>bucket.name</code>	<code>string</code>	存储桶名称
<code>bucket.creationDate</code>	<code>Date</code>	存储桶创建时间。

示例

```
minioClient.listBuckets(function(err, buckets) {
  if (err) return console.log(err)
  console.log('buckets :', buckets)
})
```

bucketExists(bucketName[, callback])

验证存储桶是否存在。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>callback(<code>err</code>)</code>	<code>function</code>	如果存储桶存在的话 <code>err</code> 就是 <code>null</code> ，否则 <code>err.code</code> 是 <code>NoSuchBucket</code> 。如果没有传 <code>callback</code> 的话，则返回一个 <code>Promise</code> 对象。

示例

```

minioClient.bucketExists('mybucket', function(err) {
  if (err) {
    if (err.code == 'NoSuchBucket') return console.log("bucket does not exist.")
    return console.log(err)
  }
  // if err is null it indicates that the bucket exists.
  console.log('Bucket exists.')
})

```

removeBucket(bucketName[, callback])

删除存储桶。

参数

参数	类型	描述
bucketName	string	存储桶名称。
callback(err)	function	如果存储桶删除成功则 err 为 null。如果没有传callback的话，则返回一个 Promise 对象。

示例

```

minioClient.removeBucket('mybucket', function(err) {
  if (err) return console.log('unable to remove bucket.')
  console.log('Bucket removed successfully.')
})

```

listObjects(bucketName, prefix, recursive)

列出存储桶中所有对象。

参数

参数	类型	描述
bucketName	string	存储桶名称。
prefix	string	要列出的对象的前缀(可选，默认值是 '')。
recursive	bool	true 代表递归查找， false 代表类似文件夹查找，以'/'分隔，不查子文件夹。(可选，默认值是 false)

返回值

参数	类型	描述
stream	Stream	存储桶中对象信息的流。

对象的格式如下：

参数	类型	描述
obj.name	string	对象名称。
obj.prefix	string	对象名称的前缀。
obj.size	number	对象的大小。
obj.etag	string	对象的etag值。
obj.lastModified	Date	最后修改时间。

示例

```
var stream = minioClient.listObjects('mybucket','', true)
stream.on('data', function(obj) { console.log(obj) })
stream.on('error', function(err) { console.log(err) })
```

listObjectsV2(bucketName, prefix, recursive)

使用S3 listing objects V2版本API列出所有对象。

参数

参数	类型	描述
bucketName	string	存储桶名称。
prefix	string	要列出的对象的前缀。（可选，默认值是 ''）
recursive	bool	true 代表递归查找， false 代表类似文件夹查找，以'/'分隔，不查子文件夹。（可选， 默认值是 false）

返回值

参数	类型	描述
stream	Stream	存储桶中对象信息的流。

对象的格式如下：

参数	类型	描述
obj.name	string	对象名称。
obj.prefix	string	对象名称的前缀。
obj.size	number	对象的大小。
obj.etag	string	对象的etag值。
obj.lastModified	Date	最后修改时间。

示例

```
var stream = minioClient.listObjectsV2('mybucket','', true)
stream.on('data', function(obj) { console.log(obj) })
stream.on('error', function(err) { console.log(err) })
```

listIncompleteUploads(bucketName, prefix, recursive)

列出存储桶中未完整上传的对象。

参数

参数	类型	描述
bucketname	string	存储桶名称。
prefix	string	未完整上传的对象的前缀。（可选，默认值是 ''）。
recursive	bool	true 代表递归查找， false 代表类似文件夹查找，以'/'分隔，不查子文件夹。（可选， 默认值是 false）

返回值

参数	类型	描述

stream

Stream

对象格式如下：

参数	类型	描述
part.key	string	对象名称。
part.uploadId	string	对象的上传ID。
part.size	Integer	未完整上传的对象的大小。

示例

```
var Stream = minioClient.listIncompleteUploads('mybucket', '', true)
Stream.on('data', function(obj) {
  console.log(obj)
})
Stream.on('end', function() {
  console.log('End')
})
Stream.on('error', function(err) {
  console.log(err)
})
```

3. 操作对象

getObject(bucketName, objectName[, callback])

下载对象。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	对象名称。
callback(err, stream)	function	回调函数，第一个参数是错误信息。stream 是对象的内容。如果没有传callback的话，则返回一个 Promise 对象。

示例

```
var size = 0
minioClient.getObject('mybucket', 'photo.jpg', function(err, dataStream) {
  if (err) {
    return console.log(err)
  }
  dataStream.on('data', function(chunk) {
    size += chunk.length
  })
  dataStream.on('end', function() {
    console.log('End. Total size = ' + size)
  })
  dataStream.on('error', function(err) {
    console.log(err)
  })
})
```

getPartialObject(bucketName, objectName, offset, length[, callback])

下载对象中指定区间的字节数组，并返回流。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	对象名称。
offset	number	offset 是从第几个字节始
length	number	length 是要下载的字节数组长度（可选值，如果为空的话则代表从offset一直到文件的末尾）。
callback(err, stream)	function	回调函数，第一个参数是错误信息。stream 是对象的内容。如果没有传callback的话，则返回一个Promise 对象。

示例

```
var size = 0
// reads 30 bytes from the offset 10.
minioClient.getPartialObject('mybucket', 'photo.jpg', 10, 30, function(err, dataStream) {
  if (err) {
    return console.log(err)
  }
  dataStream.on('data', function(chunk) {
    size += chunk.length
  })
  dataStream.on('end', function() {
    console.log('End. Total size = ' + size)
  })
  dataStream.on('error', function(err) {
    console.log(err)
  })
})
```

fGetObject(bucketName, objectName, filePath[, callback])

下载并将对象保存成本地文件。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	对象名称。
filePath	string	要写入的本地文件路径。
callback(err)	function	如果报错的话，则会调用回调函数，传入 err 参数。如果没有传callback的话，则返回一个Promise 对象。

示例

```
var size = 0
minioClient.fGetObject('mybucket', 'photo.jpg', '/tmp/photo.jpg', function(err) {
  if (err) {
    return console.log(err)
  }
  console.log('success')
})
```

putObject(bucketName, objectName, stream, size, contentType[, callback])

从一个stream/Buffer中上传一个对象。

从**stream**中上传

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>objectName</code>	<code>string</code>	对象名称。
<code>stream</code>	<code>Stream</code>	可以读的流。
<code>size</code>	<code>number</code>	对象的大小（可选）。
<code>contentType</code>	<code>string</code>	对象的Content-Type（可选，默认是 <code>application/octet-stream</code> ）。
<code>callback(err, etag)</code>	<code>function</code>	如果 <code>err</code> 不是null则代表有错误， <code>etag string</code> 是上传的对象的 <code>etag</code> 值。如果没有传 <code>callback</code> 的话，则返回一个 <code>Promise</code> 对象。

示例

单个对象的最大大小限制在5TB。`putObject`在对象大于5MiB时，自动使用multiple parts方式上传。这样的话，当上传失败的时候，客户端只需要上传未成功的部分即可（类似断点上传）。上传的对象使用MD5SUM签名进行完整性验证。

```
var Fs = require('fs')
var file = '/tmp/40mbfile'
var fileStream = Fs.createReadStream(file)
var fileStat = Fs.stat(file, function(err, stats) {
  if (err) {
    return console.log(err)
  }
  minioClient.putObject('mybucket', '40mbfile', fileStream, stats.size, function(err, etag) {
    return console.log(err, etag) // err should be null
  })
})
```

从"Buffer"或者"string"上传

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>objectName</code>	<code>string</code>	对象名称。
<code>string or Buffer</code>	<code>Stream or Buffer</code>	字符串或者缓冲区
<code>contentType</code>	<code>string</code>	对象的Content-Type（可选，默认是 <code>application/octet-stream</code> ）。
<code>callback(err, etag)</code>	<code>function</code>	如果 <code>err</code> 不是null则代表有错误， <code>etag string</code> 是上传的对象的 <code>etag</code> 值。

示例

```
var buffer = 'Hello World'
minioClient.putObject('mybucket', 'hello-file', buffer, function(err, etag) {
  return console.log(err, etag) // err should be null
})
```

fPutObject(bucketName, objectName, filePath, contentType[, callback])

上传文件。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	对象名称。
filePath	string	要上传的文件路径。
contentType	string	对象的Content-Type。
callback(err, etag)	function	如果 err 不是null则代表有错误， etag string是上传的对象的etag值。如果没有传callback的话，则返回一个 Promise 对象。

示例

```
var file = '/tmp/40mbfile'
minioClient.fPutObject('mybucket', '40mbfile', file, 'application/octet-stream', function(err, etag) {
    return console.log(err, etag) // err should be null
})
```

Object(bucketName, objectName, sourceObject, conditions[, callback])

将源对象拷贝到指定存储桶的新对象中。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	对象名称。
sourceObject	string	源对象的名称
conditions	Conditions	允许拷贝需要满足的条件。
callback(err, {etag, lastModified})	function	如果 err 不是null则代表有错误， etag string是上传的对象的etag值，lastModified Date是新拷贝对象的最后修改时间。如果没有传callback的话，则返回一个 Promise 对象。

示例

```
var conds = new Minio.Conditions()
conds.setMatchETag('bd891862ea3e22c93ed53a098218791d')
minioClient.Object('mybucket', 'newobject', '/mybucket/srcobject', conds, function(e, data) {
    if (e) {
        return console.log(e)
    }
    console.log("Successfully copied the object:")
    console.log("etag = " + data.etag + ", lastModified = " + data.lastModified)
})
```

statObject(bucketName, objectName[, callback])

获取对象的元数据。

参数

参数	类型	描述
bucketName	string	存储桶名称。

<code>objectName</code>	<code>string</code>	对象名称。
<code>callback(err, stat)</code>	<code>function</code>	如果 <code>err</code> 不是null则代表有错误, <code>stat</code> 含有对象的元数据信息, 格式如下所示。如果没有传 <code>callback</code> 的话, 则返回一个 <code>Promise</code> 对象。

参数	类型	描述
<code>stat.size</code>	<code>number</code>	对象的大小。
<code>stat.etag</code>	<code>string</code>	对象的 <code>etag</code> 值。
<code>stat.contentType</code>	<code>string</code>	对象的Content-Type。
<code>stat.lastModified</code>	<code>string</code>	Last 最后修改时间。

示例

```
minioClient.statObject('mybucket', 'photo.jpg', function(err, stat) {
  if (err) {
    return console.log(err)
  }
  console.log(stat)
})
```

removeObject(bucketName, objectName[, callback])

删除一个对象。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>objectName</code>	<code>string</code>	对象名称。
<code>callback(err)</code>	<code>function</code>	如果 <code>err</code> 不是null则代表有错误。如果没有传 <code>callback</code> 的话, 则返回一个 <code>Promise</code> 对象。

示例

```
minioClient removeObject('mybucket', 'photo.jpg', function(err) {
  if (err) {
    return console.log('Unable to remove object', err)
  }
  console.log('Removed the object')
})
```

removeIncompleteUpload(bucketName, objectName[, callback])

删除一个未完整上传的对象。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>objectName</code>	<code>string</code>	对象名称。
<code>callback(err)</code>	<code>function</code>	如果 <code>err</code> 不是null则代表有错误。如果没有传 <code>callback</code> 的话, 则返回一个 <code>Promise</code> 对象。

示例

```

minioClient.removeIncompleteUpload('mybucket', 'photo.jpg', function(err) {
  if (err) {
    return console.log('Unable to remove incomplete object', err)
  }
  console.log('Incomplete object removed successfully.')
})

```

4. Presigned操作

Presigned URLs用于对私有对象提供临时的上传/下载功能。

presignedUrl(httpMethod, bucketName, objectName, expiry[, reqParams, cb])

生成一个给指定HTTP方法（'httpMethod'）请求用的presigned URL。浏览器/移动端的客户端可以用这个URL进行下载，即使其所在的存储桶是私有的。这个presigned URL可以设置一个失效时间，默认值是7天。

参数

参数	类型	描述
httpMethod	string	http方法，put、get等。
bucketName	string	存储桶名称。
objectName	string	对象名称。
expiry	number	失效时间（以秒为单位），默认是7天，不得大于七天。
reqParams	object	请求参数。
callback(err, presignedUrl)	function	如果 err 不是null则代表有错误。presignedUrl 就是可临时上传/下载文件的 URL。如果没有传callback的话，则返回一个 Promise 对象。

示例1

```

// presigned url for 'getObject' method.
// expires in a day.
minioClient.presignedUrl('GET', 'mybucket', 'hello.txt', 24*60*60, function(err, presignedUrl) {
  if (err) return console.log(err)
  console.log(presignedUrl)
})

```

示例2

```

// presigned url for 'listObject' method.
// Lists objects in 'myBucket' with prefix 'data'.
// Lists max 1000 of them.
minioClient.presignedUrl('GET', 'mybucket', '', 1000, {'prefix': 'data', 'max-keys': 1000}, function(err, presignedUrl) {
  if (err) return console.log(err)
  console.log(presignedUrl)
})

```

presignedGetObject(bucketName, objectName, expiry[, cb])

生成一个给HTTP GET请求用的presigned URL。浏览器/移动端的客户端可以用这个URL进行下载，即使其所在的存储桶是私有的。这个presigned URL可以设置一个失效时间，默认值是7天。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	对象名称。
expiry	number	失效时间（以秒为单位），默认是7天，不得大于七天。
callback(err, presignedUrl)	function	如果 err 不是null则代表有错误。 presignedUrl 就是可用于临时下载的URL。 如果没有传callback的话，则返回一个 Promise 对象。

示例

```
// expires in a day.
minioClient.presignedGetObject('mybucket', 'hello.txt', 24*60*60, function(err, presignedUrl) {
  if (err) return console.log(err)
  console.log(presignedUrl)
})
```

presignedPutObject(bucketName, objectName, expiry[, callback])

生成一个给HTTP PUT请求用的presigned URL。浏览器/移动端的客户端可以用这个URL进行上传，即使其所在的存储桶是私有的。这个presigned URL可以设置一个失效时间，默认值是7天。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	对象名称。
expiry	number	失效时间（以秒为单位），默认是7天，不得大于七天。
callback(err, presignedUrl)	function	如果 err 不是null则代表有错误。 presignedUrl 用于使用PUT请求进行上传。如果没有传callback的话，则返回一个 Promise 对象。

示例

```
// expires in a day.
minioClient.presignedPutObject('mybucket', 'hello.txt', 24*60*60, function(err, presignedUrl) {
  if (err) return console.log(err)
  console.log(presignedUrl)
})
```

presignedPostPolicy(policy[, callback])

允许给POST请求的presigned URL设置条件策略。比如接收上传的存储桶名称、名称前缀、过期策略。

参数

参数	类型	描述
policy	object	通过minioClient.newPostPolicy()创建的Policy对象。
callback(err, {postURL, formData})	function	如果 err 不是null则代表有错误。 postURL 用于使用post请求上传。 formData 是 POST请求体中的键值对对象。如果没有传callback的话，则返回一个 Promise 对象。

创建策略：

```
var policy = minioClient.newPostPolicy()
```

设置上传策略：

```
// Policy restricted only for bucket 'mybucket'.
policy.setBucket('mybucket')

// Policy restricted only for hello.txt object.
policy.setKey('hello.txt')
```

或者

```
// Policy restricted for incoming objects with keyPrefix.
policy.setKeyStartsWith('keyPrefix')

var expires = new Date
expires.setSeconds(24 * 60 * 60 * 10)
// Policy expires in 10 days.
policy.setExpires(expires)

// Only allow 'text'.
policy.setContentType('text/plain')

// Only allow content size in range 1KB to 1MB.
policy.setContentLengthRange(1024, 1024*1024)
```

使用 `superagent` 通过浏览器POST你的数据:

```
minioClient.presignedPostPolicy(policy, function(err, data) {
  if (err) return console.log(err)

  var req = superagent.post(data.postURL)
  _.each(data.formData, function(value, key) {
    req.field(key, value)
  })

  // file contents.
  req.attach('file', '/path/to/hello.txt', 'hello.txt')

  req.end(function(err, res) {
    if (err) {
      return console.log(err.toString())
    }
    console.log('Upload successful.')
  })
})
```

5. 存储桶策略/通知

存储桶可以配置在指定事件类型和相应路径上触发通知。

getBucketNotification(bucketName[, cb])

获取指定存储桶名称的通知配置。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
		如果 <code>err</code> 不是null则代表有错误。 <code>bucketNotificationConfig</code> 是相应存

<code>bucketNotificationConfig</code>	<code>function</code>	储桶上的通知配置对象。如果没有传callback的话，则返回一个 <code>Promise</code> 对象。
---------------------------------------	-----------------------	---

示例

```
minioClient.getBucketNotification('mybucket', function(err, bucketNotificationConfig) {
  if (err) return console.log(err)
  console.log(bucketNotificationConfig)
})
```

setBucketNotification(bucketName, bucketNotificationConfig[, callback])

上传一个用户创建的通知配置，并绑定到指定的存储桶上。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>bucketNotificationConfig</code>	<code>BucketNotification</code>	包含通知配置的Javascript对象。
<code>callback(err)</code>	<code>function</code>	如果 <code>err</code> 不是null则代表有错误。如果没有传callback的话，则返回一个 <code>Promise</code> 对象。

示例

```
// Create a new notification object
var bucketNotification = new Minio.NotificationConfig();

// Setup a new Queue configuration
var arn = Minio.buildARN('aws', 'sns', 'us-west-2', '1', 'webhook')
var queue = new Minio.QueueConfig(arn)
queue.addFilterSuffix('.jpg')
queue.addFilterPrefix('myphotos/')
queue.addEvent(Minio.ObjectReducedRedundancyLostObject)
queue.addEvent(Minio.ObjectCreatedAll)

// Add the queue to the overall notification object
bucketNotification.add(queue)

minioClient.setBucketNotification('mybucket', bucketNotification, function(err) {
  if (err) return console.log(err)
  console.log('Success')
})
```

removeAllBucketNotification(bucketName[, callback])

删除指定存储桶上的通知配置。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>callback(err)</code>	<code>function</code>	如果 <code>err</code> 不是null则代表有错误。如果没有传callback的话，则返回一个 <code>Promise</code> 对象。

```
minioClient.removeAllBucketNotification('my-bucketname', function(e) {
  if (e) {
```

```

        return console.log(e)
    }
    console.log("True")
})

```

listenBucketNotification(bucketName, prefix, suffix, events)

监听存储桶上的通知，可通过前缀、后缀、事件类型进行过滤。使用本API并不需要预先设置存储桶通知。这是Minio的一个扩展API，服务端基于过来的请求使用唯一ID自动注册或者取消注册。

返回一个 `EventEmitter` 对象，它可以广播一个 `通知` 事件。

停止监听，调用 `EventEmitter` 的 `stop()` 方法。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>prefix</code>	<code>string</code>	用于过滤通知的对象名称前缀。
<code>suffix</code>	<code>string</code>	用于过滤通知的对象名称后缀。
<code>events</code>	<code>Array</code>	在指定事件类型上开启通知。

这里是你要的[完整示例](#)，拿走不谢。

```

var listener = minioClient.listenBucketNotification('my-bucketname', 'photos/', '.jpg', ['s3:ObjectCreated:*'])
listener.on('notification', function(record) {
    // For example: 's3:ObjectCreated:Put event occurred (2016-08-23T18:26:07.214Z)'
    console.log(`%s event occurred (%s)`, record.eventName, record.eventTime)
    listener.stop()
})

```

getBucketPolicy(bucketName, objectPrefix[, callback])

获取指定存储桶的访问策略，如果 `objectPrefix` 不为空，则会取相应用对象前缀上的访问策略。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>objectPrefix</code>	<code>string</code>	用于过滤的对象前缀， <code>''</code> 代表整个存储桶。
<code>callback(err, policy)</code>	<code>function</code>	如果 <code>err</code> 不是null则代表有错误。 <code>policy</code> 是存储桶策略的字符串表示 (<code>minio.Policy.NONE</code> ， <code>minio.Policy.READONLY</code> ， <code>minio.Policy.WRITEONLY</code> ，或者 <code>minio.Policy.READWRITE</code>)。如果没有传 <code>callback</code> 的话，则返回一个 <code>Promise</code> 对象。

```

// Retrieve bucket policy of 'my-bucketname' that applies to all objects that
// start with 'img-'.
minioClient.getBucketPolicy('my-bucketname', 'img-', function(err, policy) {
    if (err) throw err

    console.log(`Bucket policy: ${policy}`)
})

```

setBucketPolicy(bucketName, objectPrefix, bucketPolicy[, callback])

设置指定存储桶的策略。如果 `objectPrefix` 不为空，则会给符合该前缀的对象（们）设置策略。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>objectPrefix</code>	<code>string</code>	要设置访问策略的对象前缀。`''`代表整个存储桶。
<code>bucketPolicy</code>	<code>string</code>	存储桶策略。可选值有： <code>minio.Policy.NONE</code> ， <code>minio.Policy.READONLY</code> ， <code>minio.Policy.WRITEONLY</code> 或者 <code>minio.Policy.READWRITE</code> 。
<code>callback(err)</code>	<code>function</code>	如果 <code>err</code> 不是 <code>null</code> 则代表有错误。如果没有传 <code>callback</code> 的话，则返回一个 <code>Promise</code> 对象。

```
// Set the bucket policy of `my-bucketname` to `readonly` (only allow retrieval),
// but only for objects that start with 'img-'.
minioClient.setBucketPolicy('my-bucketname', 'img-', minio.Policy.READONLY, function(err) {
  if (err) throw err

  console.log('Set bucket policy to \'readonly\'.')
})
```

6. 了解更多

- [创建属于你的购物APP示例](#)

适用于与Amazon S3兼容的云存储的MinIO Java SDK

MinIO Java Client SDK提供简单的API来访问任何与Amazon S3兼容的对象存储服务。

本快速入门指南将向你展示如何安装客户端SDK并执行示例java程序。有关API和示例的完整列表，请查看[Java Client API Reference](#)文档。

最低需求

Java 1.8或更高版本:

- OracleJDK 8.0
- OpenJDK8.0

使用maven

```
<dependency>
    <groupId>io.minio</groupId>
    <artifactId>minio</artifactId>
    <version>7.0.2</version>
</dependency>
```

使用gradle

```
dependencies {
    compile 'io.minio:minio:7.0.2'
}
```

直接下载JAR

你可以到[maven仓库](#)直接下载最新版的JAR。

快速入门示例—文件上传

本示例程序连接到一个对象存储服务，创建一个存储桶并上传一个文件到该桶中。

你需要有存储服务的三个参数才能连接到该服务。

参数	说明
Endpoint	对象存储服务的URL
Access Key	Access key就像用户ID，可以唯一标识你的账户。
Secret Key	Secret key是你账户的密码。

在下面的例子中，我们将使用一个运行在<https://play.min.io>的免费托管的MinIO服务。你可以随意使用此服务进行测试和开发。此示例中显示的访问凭据是公开的。

FileUploader.java

```
import java.io.IOException;
import java.security.NoSuchAlgorithmException;
import java.security.InvalidKeyException;
```

```

import org.xmlpull.v1.XmlPullParserException;

import io.minio.MinioClient;
import io.minio.errors.MinioException;

public class FileUploader {
    public static void main(String[] args) throws NoSuchAlgorithmException, IOException, InvalidKeyException, XmlPullParserException {
        try {
            // 使用MinIO服务的URL, 端口, Access key和Secret key创建一个MinioClient对象
            MinioClient minioClient = new MinioClient("https://play.min.io", "Q3AM3UQ867SPQQA43P2F", "zuf+tfteSlswr
u7BJ86wekitnifILbZam1KYY3TG");

            // 检查存储桶是否已经存在
            boolean isExist = minioClient.bucketExists("asiatrip");
            if(isExist) {
                System.out.println("Bucket already exists.");
            } else {
                // 创建一个名为asiatrip的存储桶, 用于存储照片的zip文件。
                minioClient.makeBucket("asiatrip");
            }

            // 使用putObject上传一个文件到存储桶中。
            minioClient.putObject("asiatrip", "asiaphotos.zip", "/home/user/Photos/asiaphotos.zip");
            System.out.println("/home/user/Photos/asiaphotos.zip is successfully uploaded as asiaphotos.zip to `asi
atrip` bucket.");
            } catch(MinioException e) {
                System.out.println("Error occurred: " + e);
            }
        }
    }
}

```

编译FileUploader

```
javac -cp "minio-3.0.9-all.jar" FileUploader.java
```

运行FileUploader

```

java -cp "minio-3.0.9-all.jar:." FileUploader
/home/user/Photos/asiaphotos.zip is successfully uploaded as asiaphotos.zip to `asiatrip` bucket.

mc ls play/asiatrip/
[2016-06-02 18:10:29 PDT] 82KiB asiaphotos.zip

```

API文档

下面链接是完整的API文档

- [API完整文档](#)

API文档: 操作存储桶

- [makeBucket](#)
- [listBuckets](#)
- [bucketExists](#)
- [removeBucket](#)

- `listObjects`
- `listIncompleteUploads`

API文档: 操作文件对象

- `getObject`
- `putObject`
- `Object`
- `statObject`
- `removeObject`
- `removeIncompleteUpload`

API文档: Presigned操作

- `presignedGetObject`
- `presignedPutObject`
- `presignedPostPolicy`

API文档: 操作存储桶策略

- `getBucketPolicy`
- `setBucketPolicy`

完整示例

完整示例: Bucket Operations

- `ListBuckets.java`
- `ListObjects.java`
- `BucketExists.java`
- `MakeBucket.java`
- `RemoveBucket.java`
- `ListIncompleteUploads.java`

完整示例: Object Operations

- `PutObject.java`
- `PutObjectEncrypted.java`
- `GetObject.java`
- `GetObjectEncrypted.java`
- `GetPartialObject.java`
- `RemoveObject.java`
- `RemoveObjects.java`
- `StatObject.java`

完整示例: Presigned Operations

- `PresignedGetObject.java`
- `PresignedPutObject.java`
- `PresignedPostPolicy.java`

完整示例: Bucket Policy Operations

- `SetBucketPolicy.java`
- `GetBucketPolicy.java`

了解更多

- [MinIO官方文档](#)
- [MinIO Java Client SDK API文档](#)
- [创建属于你的照片API服务-完整示例](#)

贡献

[贡献者指南](#)

Java Client API参考文档

初始化**Minio Client object**。

MinIO

```
MinioClient minioClient = new MinioClient("https://play.min.io", "Q3AM3UQ867SPQQA43P2F", "zuf+tfteS1swRu7BJ86  
wekitnifILbZam1KYY3TG");
```

AWS S3

```
MinioClient s3Client = new MinioClient("https://s3.amazonaws.com", "YOUR-ACCESSKEYID", "YOUR-SECRETACCESSKEY");
```

存储桶操作	文件对象操作	Presigned操作	存储桶策略
makeBucket	getObject	presignedGetObject	getBucketPolicy
listBuckets	putObject	presignedPutObject	setBucketPolicy
bucketExists	Object	presignedPostPolicy	
removeBucket	statObject		
listObjects	removeObject		
listIncompleteUploads	removeIncompleteUpload		

1. 构造函数

```
public MinioClient(String endpoint) throws NullPointerException, InvalidEndpointException,  
InvalidPortException
```

使用给定的**endpoint**以及匿名方式创建一个**Minio client**对象。

[查看 Javadoc](#)

```
public MinioClient(URL url) throws NullPointerException, InvalidEndpointException, InvalidPortException
```

使用给定的**url**以及匿名方式创建一个**Minio client**对象。

[查看 Javadoc](#)

```
public MinioClient(com.squareup.okhttp.HttpUrl url) throws NullPointerException, InvalidEndpointException,  
InvalidPortException
```

使用给定的**HttpUrl**以及匿名方式创建一个**Minio client**对象。

[查看 Javadoc](#)

```
public MinioClient(String endpoint, String accessKey, String secretKey) throws NullPointerException,  
InvalidEndpointException, InvalidPortException
```

使用给定的**endpoint**、**access key**和**secret key**创建一个**Minio client**对象。

[查看 Javadoc](#)

```
public MinioClient(String endpoint, int port, String accessKey, String secretKey) throws  
NullPointerException, InvalidEndpointException, InvalidPortException
```

使用给定的endpoint、port、access key和secret key创建一个Minio client对象。

[查看 Javadoc](#)

```
public MinioClient(String endpoint, String accessKey, String secretKey, boolean secure) throws  
NullPointerException, InvalidEndpointException, InvalidPortException
```

使用给定的endpoint、access key、secret key和一个secure选项（是否使用https）创建一个Minio client对象。

[查看 Javadoc](#)

```
public MinioClient(String endpoint, int port, String accessKey, String secretKey, boolean secure) throws  
NullPointerException, InvalidEndpointException, InvalidPortException
```

使用给定的endpoint、port、access key、secret key和一个secure选项（是否使用https）创建一个Minio client对象。

[查看 Javadoc](#)

```
public MinioClient(com.squareup.okhttp.HttpUrl url, String accessKey, String secretKey) throws  
NullPointerException, InvalidEndpointException, InvalidPortException
```

使用给定的HttpUrl对象、access key、secret key创建一个Minio client对象。

[查看 Javadoc](#)

```
public MinioClient(URL url, String accessKey, String secretKey) throws NullPointerException,  
InvalidEndpointException, InvalidPortException
```

使用给定的URL对象、access key、secret key创建一个Minio client对象。

[查看 Javadoc](#)

参数

参数	类型	描述
endpoint	string	endPoint是一个URL，域名，IPv4或者IPv6地址。以下是合法的endpoints: https://s3.amazonaws.com https://play.min.io localhost play.min.io
port	int	TCP/IP端口号。可选，默认值是，如果是http，则默认80端口，如果是https，则默认是443端口。
accessKey	string	accessKey类似于用户ID，用于唯一标识你的账户。
secretKey	string	secretKey是你账户的密码。
secure	boolean	如果是true，则用的是https而不是http，默认值是true。
url	URL	Endpoint URL对象。
url	HttpURL	Endpoint HttpUrl对象。

示例

MinIO

```
// 1. public MinioClient(String endpoint)  
MinioClient minioClient = new MinioClient("https://play.min.io");
```

```

// 2. public MinioClient(URL url)
MinioClient minioClient = new MinioClient(new URL("https://play.min.io"));

// 3. public MinioClient(com.squareup.okhttp.HttpUrl url)
MinioClient minioClient = new MinioClient(new HttpUrl.parse("https://play.min.io"));

// 4. public MinioClient(String endpoint, String accessKey, String secretKey)
MinioClient minioClient = new MinioClient("https://play.min.io", "Q3AM3UQ867SPQQA43P2F", "zuf+tfteSlsRu7Bj86
wekitnifILbZam1KYY3TG");

// 5. public MinioClient(String endpoint, int port, String accessKey, String secretKey)
MinioClient minioClient = new MinioClient("https://play.min.io", 9000, "Q3AM3UQ867SPQQA43P2F", "zuf+tfteSlsRu7Bj86
wekitnifILbZam1KYY3TG");

// 6. public MinioClient(String endpoint, String accessKey, String secretKey, boolean insecure)
MinioClient minioClient = new MinioClient("https://play.min.io", "Q3AM3UQ867SPQQA43P2F", "zuf+tfteSlsRu7Bj86
wekitnifILbZam1KYY3TG", true);

// 7. public MinioClient(String endpoint, int port, String accessKey, String secretKey, boolean insecure)
MinioClient minioClient = new MinioClient("https://play.min.io", 9000, "Q3AM3UQ867SPQQA43P2F", "zuf+tfteSlsRu7Bj86
wekitnifILbZam1KYY3TG", true);

// 8. public MinioClient(com.squareup.okhttp.HttpUrl url, String accessKey, String secretKey)
MinioClient minioClient = new MinioClient(new URL("https://play.min.io"), "Q3AM3UQ867SPQQA43P2F", "zuf+tfteS
lsRu7Bj86wekitnifILbZam1KYY3TG");

// 9. public MinioClient(URL url, String accessKey, String secretKey)
MinioClient minioClient = new MinioClient(HttpUrl.parse("https://play.min.io"), "Q3AM3UQ867SPQQA43P2F", "zuf+
tfteSlsRu7Bj86wekitnifILbZam1KYY3TG");

```

AWS S3

```

// 1. public MinioClient(String endpoint)
MinioClient s3Client = new MinioClient("https://s3.amazonaws.com");

// 2. public MinioClient(URL url)
MinioClient minioClient = new MinioClient(new URL("https://s3.amazonaws.com"));

// 3. public MinioClient(com.squareup.okhttp.HttpUrl url)
MinioClient s3Client = new MinioClient(new HttpUrl.parse("https://s3.amazonaws.com"));

// 4. public MinioClient(String endpoint, String accessKey, String secretKey)
MinioClient s3Client = new MinioClient("s3.amazonaws.com", "YOUR-ACCESSKEYID", "YOUR-SECRETACCESSKEY");

// 5. public MinioClient(String endpoint, int port, String accessKey, String secretKey)
MinioClient s3Client = new MinioClient("s3.amazonaws.com", 80, "YOUR-ACCESSKEYID", "YOUR-SECRETACCESSKEY");

// 6. public MinioClient(String endpoint, String accessKey, String secretKey, boolean insecure)
MinioClient s3Client = new MinioClient("s3.amazonaws.com", "YOUR-ACCESSKEYID", "YOUR-SECRETACCESSKEY", false)
;

// 7. public MinioClient(String endpoint, int port, String accessKey, String secretKey, boolean insecure)
MinioClient s3Client = new MinioClient("s3.amazonaws.com", 80, "YOUR-ACCESSKEYID", "YOUR-SECRETACCESSKEY", fal
se);

// 8. public MinioClient(com.squareup.okhttp.HttpUrl url, String accessKey, String secretKey)
MinioClient s3Client = new MinioClient(new URL("s3.amazonaws.com"), "YOUR-ACCESSKEYID", "YOUR-SECRETACCESSKE
Y");

```

```
// 9. public MinioClient(URL url, String accessKey, String secretKey)
MinioClient s3Client = new MinioClient(HttpUrl.parse("s3.amazonaws.com"), "YOUR-ACCESSKEYID", "YOUR-SECRETACC
ESSKEY");
```

2. 存储桶操作

makeBucket(String bucketName)

```
public void makeBucket(String bucketName)
```

创建一个新的存储桶

[查看Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称

返回值类型	异常
None	异常列表:
	InvalidBucketNameException :非法的存储桶名称。
	NoResponseException :服务器无响应。
	IOException :连接异常
	org.xmlpull.v1.XmlPullParserException :解析返回的XML异常
	ErrorResponseException :执行失败
	InternalException :内部异常

示例

```
try {
    // 如存储桶不存在，创建之。
    boolean found = minioClient.bucketExists("mybucket");
    if (found) {
        System.out.println("mybucket already exists");
    } else {
        // 创建名为'my-bucketname'的存储桶。
        minioClient.makeBucket("mybucket");
        System.out.println("mybucket is created successfully");
    }
} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}
```

listBuckets()

```
public List<Bucket> listBuckets()
```

列出所有存储桶。

[查看Javadoc](#)

返回值类型	异常
List Bucket : List of bucket type.	异常列表:
	NoResponseException : 服务端无响应
	IOException : 连接异常
	org.xmlpull.v1.XmlPullParserException : 解析返回的XML异常
	ErrorResponseException : 执行失败异常
	InternalException : 内部错误

示例

```
try {
    // 列出所有存储桶
    List<Bucket> bucketList = minioClient.listBuckets();
    for (Bucket bucket : bucketList) {
        System.out.println(bucket.creationDate() + ", " + bucket.name());
    }
} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}
```

bucketExists(String bucketName)

```
public boolean bucketExists(String bucketName)
```

检查存储桶是否存在。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称

返回值值类型	异常
boolean : true if the bucket exists	异常列表:
	InvalidBucketNameException : 不合法的存储桶名称。
	NoResponseException : 服务器无响应。
	IOException : 连接异常。
	org.xmlpull.v1.XmlPullParserException : 解析返回的XML异常。
	ErrorResponseException : 执行失败异常。
	InternalException : 内部错误。

示例

```
try {
    // 检查'my-bucketname'是否存在。
    boolean found = minioClient.bucketExists("mybucket");
    if (found) {
        System.out.println("mybucket exists");
    } else {
        System.out.println("mybucket does not exist");
    }
}
```

```
        }
    } catch (MinioException e) {
        System.out.println("Error occurred: " + e);
    }
}
```

removeBucket(String bucketName)

```
public void removeBucket(String bucketName)
```

删除一个存储桶。

[查看 Javadoc](#)

注意: - `removeBucket`不会删除存储桶里的对象, 你需要通过`removeObject` API来删除它们。

参数

参数	类型	描述
<code>bucketName</code>	<code>String</code>	存储桶名称。

返回值类型	异常
<code>None</code>	异常列表:
	<code>InvalidBucketNameException</code> : 不合法的存储桶名称。
	<code>NoResponseException</code> : 服务器无响应。
	<code>IOException</code> : 连接异常。
	<code>org.xmlpull.v1.XmlPullParserException</code> : 解析返回的XML异常。
	<code>ErrorResponseException</code> : 执行失败异常。
	<code>InternalException</code> : 内部错误。

示例

```
try {
    // 删除之前先检查`my-bucket`是否存在。
    boolean found = minioClient.bucketExists("mybucket");
    if (found) {
        // 删除`my-bucketname`存储桶, 注意, 只有存储桶为空时才能删除成功。
        minioClient.removeBucket("mybucket");
        System.out.println("mybucket is removed successfully");
    } else {
        System.out.println("mybucket does not exist");
    }
} catch(MinioException e) {
    System.out.println("Error occurred: " + e);
}
```

listObjects(String bucketName, String prefix, boolean recursive, boolean useVersion1)

```
public Iterable<Result<Item>> listObjects(String bucketName, String prefix, boolean recursive, boolean useVersion1)
```

列出某个存储桶中的所有对象。

[查看Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
prefix	String	对象名称的前缀
recursive	boolean	是否递归查找, 如果是false,就模拟文件夹结构查找。
useVersion1	boolean	如果是true, 使用版本1 REST API

返回值类型	异常
Iterable<Result<Item>> :an iterator of Result Items.	None

示例

```

try {
    // 检查'mybucket'是否存在。
    boolean found = minioClient.bucketExists("mybucket");
    if (found) {
        // 列出'my-bucketname'里的对象
        Iterable<Result<Item>> myObjects = minioClient.listObjects("mybucket");
        for (Result<Item> result : myObjects) {
            Item item = result.get();
            System.out.println(item.lastModified() + ", " + item.size() + ", " + item.objectName());
        }
    } else {
        System.out.println("mybucket does not exist");
    }
} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}

```

listIncompleteUploads(String bucketName, String prefix, boolean recursive)

```
public Iterable<Result<Upload>> listIncompleteUploads(String bucketName, String prefix, boolean recursive)
```

列出存储桶中被部分上传的对象。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
prefix	String	对象名称的前缀, 列出有该前缀的对象
recursive	boolean	是否递归查找, 如果是false,就模拟文件夹结构查找。

返回值类型	异常
Iterable<Result<Upload>> : an iterator of Upload.	None

示例

```

try {
    // 检查'mybucket'是否存在。

```

```

boolean found = minioClient.bucketExists("mybucket");
if (found) {
    // 列出'mybucket'中所有未完成的multipart上传的对象。
    Iterable<Result<Upload>> myObjects = minioClient.listIncompleteUploads("mybucket");
    for (Result<Upload> result : myObjects) {
        Upload upload = result.get();
        System.out.println(upload.uploadId() + ", " + upload.objectName());
    }
} else {
    System.out.println("mybucket does not exist");
}
} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}
}

```

getBucketPolicy(String bucketName, String objectPrefix)

```
public PolicyType getBucketPolicy(String bucketName, String objectPrefix)
```

获得指定对象前缀的存储桶策略。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectPrefix	String	策略适用的对象的前缀

返回值类型	异常
PolicyType : The current bucket policy type for a given bucket and objectPrefix.	异常列表:
	InvalidBucketNameException : 不合法的存储桶名称。
	NoResponseException : 服务器无响应。
	IOException : 连接异常。
	org.xmlpull.v1.XmlPullParserException : 解析返回的 XML 异常。
	ErrorResponseException : 执行失败异常。
	InternalException : 内部错误。
	InvalidBucketNameException : 不合法的存储桶名称。
	InvalidObjectPrefixException : 不合法的对象前缀
	NoSuchAlgorithmException : 找不到相应的签名算法。
	InsufficientDataException : 在读到相应length之前就得到一个EOFException。

示例

```

try {
    System.out.println("Current policy: " + minioClient.getBucketPolicy("myBucket", "downloads"));
} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}

```

setBucketPolicy(String bucketName, String objectPrefix, PolicyType policy)

```
public void setBucketPolicy(String bucketName, String objectPrefix, PolicyType policy)
```

给一个存储桶+对象前缀设置策略。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectPrefix	String	对象前缀。
policy	PolicyType	要赋予的策略，可选值有[PolicyType.NONE, PolicyType.READ_ONLY, PolicyType.READ_WRITE, PolicyType.WRITE_ONLY]。

返回值类型	异常
None	异常列表：
	InvalidBucketNameException : 不合法的存储桶名称。
	NoResponseException : 服务器无响应。
	IOException : 连接异常。
	org.xmlpull.v1.XmlPullParserException : 解析返回的XML异常。
	ErrorResponseException : 执行失败异常。
	InternalException : 内部错误。
	InvalidBucketNameException : 不合法的存储桶名称。
	InvalidObjectPrefixException : 不合法的对象前缀
	NoSuchAlgorithmException : 找不到相应的签名算法。
	InsufficientDataException : 在读到相应length之前就得到一个EOFException。

示例

```
try {
    minioClient.setBucketPolicy("myBucket", "uploads", PolicyType.READ_ONLY);
} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}
```

3. Object operations

getObject(String bucketName, String objectName)

```
public InputStream getObject(String bucketName, String objectName, long offset)
```

以流的形式下载一个对象。

[查看 Javadoc](#)

参数

参数	类型	描述

<code>bucketName</code>	<code>String</code>	存储桶名称。
<code>objectName</code>	<code>String</code>	存储桶里的对象名称。

返回值类型	异常
<code>InputStream : InputStream containing the object data.</code>	异常列表:
	<code>InvalidBucketNameException</code> : 不合法的存储桶名称。
	<code>NoResponseException</code> : 服务器无响应。
	<code>IOException</code> : 连接异常。
	<code>org.xmlpull.v1.XmlPullParserException</code> : 解析返回的XML异常。
	<code>ErrorResponseException</code> : 执行失败异常。
	<code>InternalException</code> : 内部错误。

示例

```

try {
    // 调用statObject()来判断对象是否存在。
    // 如果不存在，statObject()抛出异常，
    // 否则则代表对象存在。
    minioClient.statObject("mybucket", "myobject");

    // 获取"myobject"的输入流。
    InputStream stream = minioClient.getObject("mybucket", "myobject");

    // 读取输入流直到EOF并打印到控制台。
    byte[] buf = new byte[16384];
    int bytesRead;
    while ((bytesRead = stream.read(buf, 0, buf.length)) >= 0) {
        System.out.println(new String(buf, 0, bytesRead));
    }

    // 关闭流，此处为示例，流关闭最好放在finally块。
    stream.close();
} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}
}

```

getObject(String bucketName, String objectName, long offset, Long length)

```
public InputStream getObject(String bucketName, String objectName, long offset, Long length)
```

下载对象指定区域的字节数组做为流。（断点下载）

[查看 Javadoc](#)

参数

参数	类型	描述		
<code>bucketName</code>	<code>String</code>	存储桶名称。		
<code>objectName</code>	<code>String</code>	存储桶里的对象名称。		
<code>offset</code>	<code>Long</code>	<code>offset</code> 是起始字节的位置		
<code>length</code>	<code>Long</code>	<code>length</code> 是要读取的长度 (可选，如果无值则代表读到文件结尾)。		

返回值类型	异常
<code>InputStream : InputStream containing the object's data.</code>	异常列表:
	<code>InvalidBucketNameException</code> : 不合法的存储桶名称。
	<code>NoResponseException</code> : 服务器无响应。
	<code>IOException</code> : 连接异常。
	<code>org.xmlpull.v1.XmlPullParserException</code> : 解析返回的XML异常。
	<code>ErrorResponseException</code> : 执行失败异常。
	<code>InternalException</code> : 内部错误。

示例

```
try {

    // 调用statObject()来判断对象是否存在。
    // 如果不存在, statObject()抛出异常,
    // 否则则代表对象存在。
    minioClient.statObject("mybucket", "myobject");

    // 获取指定offset和length的"myobject"的输入流。
    InputStream stream = minioClient.getObject("mybucket", "myobject", 1024L, 4096L);

    // 读取输入流直到EOF并打印到控制台。
    byte[] buf = new byte[16384];
    int bytesRead;
    while ((bytesRead = stream.read(buf, 0, buf.length)) >= 0) {
        System.out.println(new String(buf, 0, bytesRead));
    }

    // 关闭流。
    stream.close();
} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}
```

getObject(String bucketName, String objectName, String fileName)

```
public void getObject(String bucketName, String objectName, String fileName)
```

下载并将文件保存到本地。

[查看 Javadoc](#)

参数

参数	类型	描述
<code>bucketName</code>	<code>String</code>	存储桶名称。
<code>objectName</code>	<code>String</code>	存储桶里的对象名称。
<code>fileName</code>	<code>String</code>	File name.

返回值类型	异常
<code>None</code>	异常列表:

	<code>InvalidBucketNameException</code> : 不合法的存储桶名称。
	<code>NoResponseException</code> : 服务器无响应。
	<code>IOException</code> : 连接异常。
	<code>org.xmlpull.v1.XmlPullParserException</code> : 解析返回的XML异常。
	<code>ErrorResponseException</code> : 执行失败异常。
	<code>InternalException</code> : 内部错误。

示例

```

try {
    // 调用statObject()来判断对象是否存在。
    // 如果不存在, statObject()抛出异常,
    // 否则则代表对象存在。
    minioClient.statObject("mybucket", "myobject");

    // 获取myobject的流并保存到photo.jpg文件中。
    minioClient.getObject("mybucket", "myobject", "photo.jpg");

} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}

```

getObject(String bucketName, String objectName, SecretKey key)

```
public CipherInputStream getObject(String bucketName, String objectName, SecretKey key)
```

在给定的存储桶中获取整个加密对象的数据作为InputStream，然后用传入的master key解密和加密对象关联的content key。然后创建一个含有InputStream和Cipher的CipherInputStream。这个Cipher被初始为用于使用content key进行解密，所以CipherInputStream会在返回数据前，尝试读取数据并进行解密。所以read()方法返回的是处理过的原始对象数据。

CipherInputStream必须用完关闭，否则连接不会被释放。

[查看 Javadoc](#)

参数

参数	类型	描述
<code>bucketName</code>	<code>String</code>	存储桶名称。
<code>objectName</code>	<code>String</code>	存储桶里的对象名称。
<code>key</code>	<code>SecretKey</code>	<code>SecretKey</code> 类型的数据。

返回值类型	异常
<code>None</code>	异常列表：
	<code>InvalidBucketNameException</code> : 不合法的存储桶名称。
	<code>NoResponseException</code> : 服务器无响应。
	<code>IOException</code> : 连接异常。
	<code>org.xmlpull.v1.XmlPullParserException</code> : 解析返回的XML异常。
	<code>ErrorResponseException</code> : 执行失败异常。
	<code>InternalException</code> : 内部错误。
	<code>InvalidEncryptionMetadataException</code> : 加密秘钥错误。

	BadPaddingException : 错误的padding
	IllegalBlockSizeException : 不正确的block size
	NoSuchPaddingException : 错误的padding类型
	InvalidAlgorithmParameterException : 该算法不存在

示例

```

try {
    // 调用statObject()来判断对象是否存在。
    // 如果不存在，statObject()抛出异常，
    // 否则则代表对象存在。
    minioClient.statObject("mybucket", "myobject");

    //生成256位AES key。
    KeyGenerator symKeyGenerator = KeyGenerator.getInstance("AES");
    symKeyGenerator.init(256);
    SecretKey symKey = symKeyGenerator.generateKey();

    // 获取对象数据并保存到photo.jpg
    InputStream stream = minioClient.getObject("testbucket", "my-objectname", symKey);

    // 读流到EOF，并输出到控制台。
    byte[] buf = new byte[16384];
    int bytesRead;
    while ((bytesRead = stream.read(buf, 0, buf.length)) >= 0) {
        System.out.println(new String(buf, 0, bytesRead, StandardCharsets.UTF_8));
    }

    // 关闭流。
    stream.close();
}

} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}

```

getObject(String bucketName, String objectName, KeyPair key)

```
public InputStream getObject(String bucketName, String objectName, KeyPair key)
```

在给定的存储桶中获取整个加密对象的数据作为InputStream，然后用传入的master keyPair解密和加密对象关联的content key。然后创建一个含有InputStream和Cipher的CipherInputStream。这个Cipher被初始为用于使用content key进行解密，所以CipherInputStream会在返回数据前，尝试读取数据并进行解密。所以read()方法返回的是处理过的原始对象数据。

CipherInputStream必须用完关闭，否则连接不会被释放。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。
key	KeyPair	RSA KeyPair类型的对象。

返回值类型	异常

None	异常列表:
	<code>InvalidBucketNameException</code> : 不合法的存储桶名称。
	<code>NoResponseException</code> : 服务器无响应。
	<code>IOException</code> : 连接异常。
	<code>org.xmlpull.v1.XmlPullParserException</code> : 解析返回的XML异常。
	<code>ErrorResponseException</code> : 执行失败异常。
	<code>InternalException</code> : 内部错误。
	<code>InvalidEncryptionMetadataException</code> : 加密秘钥错误。
	<code>BadPaddingException</code> : 错误的padding
	<code>IllegalBlockSizeException</code> : 不正确的block size
	<code>NoSuchPaddingException</code> : 错误的padding类型
	<code>InvalidAlgorithmParameterException</code> : 该算法不存在

示例

```

try {
    // 调用statObject()来判断对象是否存在。
    // 如果不存在，statObject()抛出异常，
    // 否则则代表对象存在。
    minioClient.statObject("mybucket", "myobject");

    KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("RSA");
    keyGenerator.initialize(1024, new SecureRandom());
    KeyPair keypair = keyGenerator.generateKeyPair();

    // 获取对象数据并保存到photo.jpg
    InputStream stream = minioClient.getObject("testbucket", "my-objectname", keypair);

    // 读流到EOF，并输出到控制台。
    byte[] buf = new byte[16384];
    int bytesRead;
    while ((bytesRead = stream.read(buf, 0, buf.length)) >= 0) {
        System.out.println(new String(buf, 0, bytesRead, StandardCharsets.UTF_8));
    }

    // 关闭流。
    stream.close();

} catch (MinioException e) {
    System.out.println("Error occurred: " + e);
}

```

putObject(String bucketName, String objectName, InputStream stream, long size, String contentType)

```

public void putObject(String bucketName, String objectName, InputStream stream, long size, String contentType)

```

通过InputStream上传对象。

[查看 Javadoc](#)

参数

参数	类型	描述		
bucketName	String	存储桶名称。		
objectName	String	存储桶里的对象名称。		
stream	InputStream	要上传的流。		
size	long	要上传的 stream 的 size		
contentType	String	Content type。		

返回值类型	异常
None	异常列表:
	InvalidBucketNameException : 不合法的存储桶名称。
	NoResponseException : 服务器无响应。
	IOException : 连接异常。
	org.xmlpull.v1.XmlPullParserException : 解析返回的XML异常。
	ErrorResponseException : 执行失败异常。
	InternalException : 内部错误。

示例

单个对象的最大大小限制在5TB。putObject在对象大于5MiB时，自动使用multiple parts方式上传。这样，当上传失败时，客户端只需要上传未成功的部分即可（类似断点上传）。上传的对象使用MD5SUM签名进行完整性验证。

```

try {
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < 1000; i++) {
        builder.append("Sphinx of black quartz, judge my vow: Used by Adobe InDesign to display font samples. ");
        builder.append("(29 letters)\n");
        builder.append("Jackdaws love my big sphinx of quartz: Similarly, used by Windows XP for some fonts. ");
        builder.append("(31 letters)\n");
        builder.append("Pack my box with five dozen liquor jugs: According to Wikipedia, this one is used on ");
        builder.append("NASAs Space Shuttle. (32 letters)\n");
        builder.append("The quick onyx goblin jumps over the lazy dwarf: Flavor text from an Unhinged Magic Card.");
    };
    builder.append("(39 letters)\n");
    builder.append("How razorback-jumping frogs can level six piqued gymnasts!: Not going to win any brevity ");
);
    builder.append("awards at 49 letters long, but old-time Mac users may recognize it.\n");
    builder.append("Cozy lumox gives smart squid who asks for job pen: A 41-letter tester sentence for Mac ");
);
    builder.append("computers after System 7.\n");
    builder.append("A few others we like: Amazingly few discotheques provide jukeboxes; Now fax quiz Jack! my ");
);
    builder.append("brave ghost pled; Watch Jeopardy!, Alex Trebeeks fun TV quiz game.\n");
    builder.append("- --\n");
}
ByteArrayList bais = new
ByteArrayList(builder.toString().getBytes("UTF-8"));
// 创建对象
minioClient.putObject("mybucket", "myobject", bais, bais.available(), "application/octet-stream");
bais.close();
System.out.println("myobject is uploaded successfully");
} catch(MinioException e) {
    System.out.println("Error occurred: " + e);
}

```

putObject(String bucketName, String objectName, String fileName)

```
public void putObject(String bucketName, String objectName, String fileName)
```

通过文件上传到对象中。 [查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。
fileName	String	File name.

返回值类型	异常
None	异常列表:
	InvalidBucketNameException : 不合法的存储桶名称。
	NoResponseException : 服务器无响应。
	IOException : 连接异常。
	org.xmlpull.v1.XmlPullParserException : 解析返回的XML异常。
	ErrorResponseException : 执行失败异常。
	InternalException : 内部错误。

示例

```
try {
    minioClient.putObject("mybucket", "island.jpg", "/mnt/photos/island.jpg")
    System.out.println("island.jpg is uploaded successfully");
} catch(MinioException e) {
    System.out.println("Error occurred: " + e);
}
```

putObject(String bucketName, String objectName, InputStream stream, long size, String contentType, SecretKey key)

```
public void putObject(String bucketName, String objectName, InputStream stream, long size, String contentType
, SecretKey key)
```

拿到流的数据，使用随机生成的content key进行加密，并上传到指定存储桶中。同时将加密后的content key和iv做为加密对象有header也上传到存储桶中。content key使用传入到该方法的master key进行加密。

如果对象大于5MB,客户端会自动进行multi part上传。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。
stream	InputStream	要上传的流。

<code>size</code>	<code>long</code>	要上传的流的大小。
<code>contentType</code>	<code>String</code>	Content type。
<code>key</code>	<code>SecretKey</code>	用AES初使化的对象 <code>SecretKey</code> 。

返回值类型	异常
<code>None</code>	异常列表:
	<code>InvalidBucketNameException</code> : 不合法的存储桶名称。
	<code>NoResponseException</code> : 服务器无响应。
	<code>IOException</code> : 连接异常。
	<code>org.xmlpull.v1.XmlPullParserException</code> : 解析返回的XML异常。
	<code>ErrorResponseException</code> : 执行失败异常。
	<code>InternalException</code> : 内部错误。
	<code>InvalidAlgorithmParameterException</code> : 错误的加密算法。
	<code>BadPaddingException</code> : 不正确的padding。
	<code>IllegalBlockSizeException</code> : 不正确的block。
	<code>NoSuchPaddingException</code> : 错误的padding类型。

示例

对象使用随机生成的key进行加密，然后这个用于加密数据的key又被由仅被client知道的master key(封装在`encryptionMaterials`对象里)进行加密。这个被加密后的key和IV做为对象的header和加密后的对象一起被上传到存储服务上。

```

try {
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < 1000; i++) {
        builder.append("Sphinx of black quartz, judge my vow: Used by Adobe InDesign to display font samples. ");
        builder.append("(29 letters)\n");
        builder.append("Jackdaws love my big sphinx of quartz: Similarly, used by Windows XP for some fonts. ");
        builder.append("(31 letters)\n");
        builder.append("Pack my box with five dozen liquor jugs: According to Wikipedia, this one is used on ");
        builder.append("(32 letters)\n");
        builder.append("The quick onyx goblin jumps over the lazy dwarf: Flavor text from an Unhinged Magic Card. ");
    };
    builder.append("(39 letters)\n");
    builder.append("How razorback-jumping frogs can level six piqued gymnasts!: Not going to win any brevity ");
);
    builder.append("awards at 49 letters long, but old-time Mac users may recognize it.\n");
    builder.append("Cozy lumox gives smart squid who asks for job pen: A 41-letter tester sentence for Mac ");
);
    builder.append("computers after System 7.\n");
    builder.append("A few others we like: Amazingly few discotheques provide jukeboxes; Now fax quiz Jack! my ");
);
    builder.append("brave ghost pled; Watch Jeopardy!, Alex Trebeeks fun TV quiz game.\n");
    builder.append("- --\n");
}
ByteArrayList bais = new
ByteArrayList(builder.toString().getBytes("UTF-8"));

//生成256位AES key.
KeyGenerator symKeyGenerator = KeyGenerator.getInstance("AES");
symKeyGenerator.init(256);
SecretKey symKey = symKeyGenerator.generateKey();

```

```

// 创建一个对象
minioClient.putObject("mybucket", "myobject", bais, bais.available(), "application/octet-stream", symKey);
bais.close();
System.out.println("myobject is uploaded successfully");
} catch(MinioException e) {
    System.out.println("Error occurred: " + e);
}

```



putObject(String bucketName, String objectName, InputStream stream, long size, String contentType, KeyPair key)

```

public void putObject(String bucketName, String objectName, InputStream stream, long size, String contentType
, KeyPair key)

```

拿到流的数据，使用随机生成的content key进行加密，并上传到指定存储桶中。同时将加密后的content key和iv做为加密对象有header也上传到存储桶中。content key使用传入到该方法的master key进行加密。

如果对象大于5MB,客户端会自动进行multi part上传。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。
stream	InputStream	要上传的流。
size	long	要上传的流的大小。
contentType	String	Content type。
key	KeyPair	一个RSA KeyPair的对象。

返回值类型	异常
None	异常列表：
	InvalidBucketNameException : 不合法的存储桶名称。
	NoResponseException : 服务器无响应。
	IOException : 连接异常。
	org.xmlpull.v1.XmlPullParserException : 解析返回的XML异常。
	ErrorResponseException : 执行失败异常。
	InternalException : 内部错误。
	InvalidAlgorithmParameterException : 错误的加密算法。
	BadPaddingException : 不正确的padding。
	IllegalBlockSizeException : 不正确的block。
	NoSuchPaddingException : 错误的padding类型。

示例

对象使用随机生成的key进行加密，然后这个用于加密数据的key又被由仅被client知道的master key(封装在encryptionMaterials对象里)进行加密。这个被加密后的key和IV做为对象的header和加密后的对象一起被上传到存储服务上。

```
try {
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < 1000; i++) {
        builder.append("Sphinx of black quartz, judge my vow: Used by Adobe InDesign to display font samples. ");
        builder.append("(29 letters)\n");
        builder.append("Jackdaws love my big sphinx of quartz: Similarly, used by Windows XP for some fonts. ");
        builder.append("(31 letters)\n");
        builder.append("Pack my box with five dozen liquor jugs: According to Wikipedia, this one is used on ");
        builder.append("NASAs Space Shuttle. (32 letters)\n");
        builder.append("The quick onyx goblin jumps over the lazy dwarf: Flavor text from an Unhinged Magic Card.");
    );
    builder.append("(39 letters)\n");
    builder.append("How razorback-jumping frogs can level six piqued gymnasts!: Not going to win any brevity ");
);
    builder.append("awards at 49 letters long, but old-time Mac users may recognize it.\n");
    builder.append("Cozy lumox gives smart squid who asks for job pen: A 41-letter tester sentence for Mac ");
);
    builder.append("computers after System 7.\n");
    builder.append("A few others we like: Amazingly few discotheques provide jukeboxes; Now fax quiz Jack! my ");
);
    builder.append("brave ghost pled; Watch Jeopardy!, Alex Trebeeks fun TV quiz game.\n");
    builder.append("- --\n");
}
ByteArrayInputStream bais = new
ByteArrayInputStream(builder.toString().getBytes("UTF-8"));

KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("RSA");
keyGenerator.initialize(1024, new SecureRandom());
KeyPair keypair = keyGenerator.generateKeyPair();

// Create an object
minioClient.putObject("mybucket", "myobject", bais, bais.available(), "application/octet-stream", keypair);
bais.close();
System.out.println("myobject is uploaded successfully");
} catch(MinioException e) {
    System.out.println("Error occurred: " + e);
}
```

statObject(String bucketName, String objectName)

```
public ObjectStat statObject(String bucketName, String objectName)
```

获取对象的元数据。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。

返回值类型	异常

ObjectStat : Populated object meta data.	异常列表:
	InvalidBucketNameException : 不合法的存储桶名称。
	NoResponseException : 服务器无响应。
	IOException : 连接异常。
	org.xmlpull.v1.XmlPullParserException : 解析返回的XML异常。
	ErrorResponseException : 执行失败异常。
	InternalException : 内部错误。

示例

```
try {
    // 获得对象的元数据。
    ObjectStat objectStat = minioClient.statObject("mybucket", "myobject");
    System.out.println(objectStat);
} catch(MinioException e) {
    System.out.println("Error occurred: " + e);
}
```

Object(String bucketName, String objectName, String destBucketName, String destObjectName, Conditions cpConds, Map metadata)

```
public void Object(String bucketName, String objectName, String destBucketName, String destObjectName, Conditions cpConds, Map<String, String> metadata)
```

从objectName指定的对象中将数据拷贝到destObjectName指定的对象。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	源存储桶名称。
objectName	String	源存储桶中的源对象名称。
destBucketName	String	目标存储桶名称。
destObjectName	String	要创建的目标对象名称,如果为空, 默认为源对象名称。
Conditions	Conditions	拷贝操作的一些条件Map。
metadata	Map	给目标对象的元数据Map。

返回值类型	异常
None	异常列表:
	InvalidBucketNameException : 不合法的存储桶名称。
	NoResponseException : 服务器无响应。
	IOException : 连接异常。
	org.xmlpull.v1.XmlPullParserException : 解析返回的XML异常。
	ErrorResponseException : 执行失败异常。
	InternalException : 内部错误。

示例

本API执行了一个服务端的拷贝操作。

```

try {
    Conditions Conditions = new Conditions();
    Conditions.setMatchETagNone("TestETag");

    minioClient.Object("mybucket", "island.jpg", "mydestbucket", "processed.png", Conditions);
    System.out.println("island.jpg is uploaded successfully");
} catch(MinioException e) {
    System.out.println("Error occurred: " + e);
}

```

removeObject(String bucketName, String objectName)

```
public void removeObject(String bucketName, String objectName)
```

删除一个对象。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。

返回值类型	异常
None	异常列表:
	InvalidBucketNameException : 不合法的存储桶名称。
	NoResponseException : 服务器无响应。
	IOException : 连接异常。
	org.xmlpull.v1.XmlPullParserException : 解析返回的XML异常。
	ErrorResponseException : 执行失败异常。
	InternalException : 内部错误。

示例

```

try {
    // 从mybucket中删除myobject。
    minioClient.removeObject("mybucket", "myobject");
    System.out.println("successfully removed mybucket/myobject");
} catch (MinioException e) {
    System.out.println("Error: " + e);
}

```

removeObject(String bucketName, Iterable objectNames)

```
public Iterable<Result<DeleteError>> removeObject(String bucketName, Iterable<String> objectNames)
```

删除多个对象。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectNames	Iterable	含有要删除的多个object名称的迭代器对象。

返回值类型	异常
Iterable<Result<DeleteError>> :an iterator of Result DeleteError.	None

示例

```
List<String> objectNames = new LinkedList<String>();
objectNames.add("my-objectname1");
objectNames.add("my-objectname2");
objectNames.add("my-objectname3");
try {
    // 删除my-bucketname里的多个对象
    for (Result<DeleteError> errorResult: minioClient.removeObject("my-bucketname", objectNames)) {
        DeleteError error = errorResult.get();
        System.out.println("Failed to remove '" + error.objectName() + "' . Error:" + error.message());
    }
} catch (MinioException e) {
    System.out.println("Error: " + e);
}
```

removeIncompleteUpload(String bucketName, String objectName)

```
public void removeIncompleteUpload(String bucketName, String objectName)
```

删除一个未完整上传的对象。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。

返回值类型	异常
None	异常列表:
	InvalidBucketNameException : 不合法的存储桶名称。
	NoResponseException : 服务器无响应。
	IOException : 连接异常。
	org.xmlpull.v1.XmlPullParserException : 解析返回的XML异常。
	ErrorResponseException : 执行失败异常。
	InternalException : 内部错误。

示例

```
try {
    // 从存储桶中删除名为myobject的未完整上传的对象。
    minioClient.removeIncompleteUpload("mybucket", "myobject");
    System.out.println("successfully removed all incomplete upload session of my-bucketname/my-objectname");
```

```
    } catch(MinioException e) {
        System.out.println("Error occurred: " + e);
    }
}
```

4. Presigned操作

presignedGetObject(String bucketName, String objectName, Integer expires)

```
public String presignedGetObject(String bucketName, String objectName, Integer expires)
```

生成一个给HTTP GET请求用的presigned URL。浏览器/移动端的客户端可以用这个URL进行下载，即使其所在的存储桶是私有的。这个presigned URL可以设置一个失效时间，默认值是7天。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。
expiry	Integer	失效时间（以秒为单位），默认是7天，不得大于七天。

返回值类型	异常
String : string contains URL to download the object.	异常列表：
	InvalidBucketNameException : 不合法的存储桶名称。
	InvalidKeyException : 不合法的access key或者secret key。
	IOException : 连接异常。
	NoSuchAlgorithmException : 找不到相应的签名算法。
	InvalidExpiresRangeException : presigned URL已经过期了。

示例

```
try {
    String url = minioClient.presignedGetObject("mybucket", "myobject", 60 * 60 * 24);
    System.out.println(url);
} catch(MinioException e) {
    System.out.println("Error occurred: " + e);
}
```

presignedPutObject(String bucketName, String objectName, Integer expires)

```
public String presignedPutObject(String bucketName, String objectName, Integer expires)
```

生成一个给HTTP PUT请求用的presigned URL。浏览器/移动端的客户端可以用这个URL进行上传，即使其所在的存储桶是私有的。这个presigned URL可以设置一个失效时间，默认值是7天。

[查看 Javadoc](#)

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。
expiry	Integer	失效时间（以秒为单位），默认是7天，不得大于七天。

返回值类型	异常
String : string contains URL to download the object.	异常列表：
	InvalidBucketNameException : 不合法的存储桶名称。
	InvalidKeyException : 不合法的access key或者secret key。
	IOException : 连接异常。
	NoSuchAlgorithmException : 找不到相应的签名算法。
	InvalidExpiresRangeException : presigned URL已经过期了。

示例

```
try {
    String url = minioClient.presignedPutObject("mybucket", "myobject", 60 * 60 * 24);
    System.out.println(url);
} catch(MinioException e) {
    System.out.println("Error occurred: " + e);
}
```

presignedPostPolicy(PostPolicy policy)

```
public Map<String, String> presignedPostPolicy(PostPolicy policy)
```

允许给POST请求的presigned URL设置策略，比如接收对象上传的存储桶名称的策略，key名称前缀，过期策略。

[查看 Javadoc](#)

参数

参数	类型	描述
policy	PostPolicy	对象的post策略

返回值类型	异常
Map : Map of strings to construct form-data.	异常列表：
	InvalidBucketNameException : 不合法的存储桶名称。
	InvalidKeyException : 不合法的access key或者secret key。
	IOException : 连接异常。
	NoSuchAlgorithmException : 找不到相应的签名算法。

示例

```
try {
    PostPolicy policy = new PostPolicy("mybucket", "myobject",
        DateTime.now().plusDays(7));
    policy.setContentType("image/png");
```

```
Map<String, String> formData = minioClient.presignedPostPolicy(policy);
System.out.print("curl -X POST ");
for (Map.Entry<String, String> entry : formData.entrySet()) {
    System.out.print(" -F " + entry.getKey() + "=" + entry.getValue());
}
System.out.println(" -F file=@/tmp/userpic.png https://play.min.io/mybucket");
} catch(MinioException e) {
    System.out.println("Error occurred: " + e);
```

5. 了解更多

- [创建属于你的照片API服务示例](#)
- [完整的JavaDoc](#)

适用于与Amazon S3兼容的云存储的MinIO Python Library

MinIO Python Client SDK提供简单的API来访问任何与Amazon S3兼容的对象存储服务。

本文我们将学习如何安装MinIO client SDK，并运行一个python的示例程序。对于完整的API以及示例，请参考[Python Client API Reference](#)。

本文假设你已经有一个可运行的[Python](#)开发环境。

最低要求

- Python 2.7或更高版本

使用pip安装

```
pip install minio
```

使用源码安装

```
git clone https://github.com/minio/minio-py
cd minio-py
python setup.py install
```

初始化MinIO Client

MinIO client需要以下4个参数来连接MinIO对象存储服务。

参数	描述
endpoint	对象存储服务的URL。
access_key	Access key是唯一标识你的账户的用户ID。
secret_key	Secret key是你账户的密码。
secure	true代表使用HTTPS。

```
from minio import Minio
from minio.error import ResponseError

minioClient = Minio('play.min.io',
                    access_key='Q3AM3UQ867SPQQA43P2F',
                    secret_key='zuf+tfteSlsrwu7BJ86wekitnifILbZam1KYY3TG',
                    secure=True)
```

示例-文件上传

本示例连接到一个MinIO对象存储服务，创建一个存储桶并上传一个文件到存储桶中。

我们在本示例中使用运行在<https://play.min.io>上的MinIO服务，你可以用这个服务来开发和测试。示例中的访问凭据是公开的。

file-uploader.py

```

# 引入MinIO包。
from minio import Minio
from minio.error import (ResponseError, BucketAlreadyOwnedByYou,
                         BucketAlreadyExists)

# 使用endpoint、access key和secret key来初始化minioClient对象。
minioClient = Minio('play.min.io',
                    access_key='Q3AM3UQ867SPQQA43P2F',
                    secret_key='zuf+tfteSlswRu7BJ86wekitnifILbZam1KYY3TG',
                    secure=True)

# 调用make_bucket来创建一个存储桶。
try:
    minioClient.make_bucket("maylogs", location="us-east-1")
except BucketAlreadyOwnedByYou as err:
    pass
except BucketAlreadyExists as err:
    pass
except ResponseError as err:
    raise
else:
    try:
        minioClient.fput_object('maylogs', 'pumaserver_debug.log', '/tmp/pumaserver_debug.log')
    except ResponseError as err:
        print(err)

```

Run file-uploader

```

python file_uploader.py

mc ls play/maylogs/
[2016-05-27 16:41:37 PDT] 12MiB pumaserver_debug.log

```

API文档

完整的API文档在这里。

- [完整API文档](#)

API文档：操作存储桶

- [make_bucket](#)
- [list_buckets](#)
- [bucket_exists](#)
- [remove_bucket](#)
- [list_objects](#)
- [list_objects_v2](#)
- [list_incomplete_uploads](#)

API文档：存储桶策略

- [get_bucket_policy](#)
- [set_bucket_policy](#)

API文档：存储桶通知

- `set_bucket_notification`
- `get_bucket_notification`
- `remove_all_bucket_notification`
- `listen_bucket_notification`

API文档：操作文件对象

- `fput_object`
- `fget_object`

API文档：操作对象

- `get_object`
- `put_object`
- `stat_object`
- `_object`
- `get_partial_object`
- `remove_object`
- `remove_objects`
- `remove_incomplete_upload`

API文档：Presigned操作

- `presigned_get_object`
- `presigned_put_object`
- `presigned_post_policy`

完整示例

完整示例：操作存储桶

- `make_bucket.py`
- `list_buckets.py`
- `bucket_exists.py`
- `list_objects.py`
- `remove_bucket.py`
- `list_incomplete_uploads.py`

完整示例：存储桶策略

- `set_bucket_policy.py`
- `get_bucket_policy.py`

完整示例：存储桶通知

- `set_bucket_notification.py`
- `get_bucket_notification.py`
- `remove_all_bucket_notification.py`
- `listen_bucket_notification.py`

完整示例：操作文件对象

- `fput_object.py`
- `fget_object.py`

完整示例：操作对象

- [get_object.py](#)
- [put_object.py](#)
- [stat_object.py](#)
- [_object.py](#)
- [get_partial_object.py](#)
- [remove_object.py](#)
- [remove_objects.py](#)
- [remove_incomplete_upload.py](#)

完整示例：**Presigned**操作

- [presigned_get_object.py](#)
- [presigned_put_object.py](#)
- [presigned_post_policy.py](#)

了解更多

- [完整文档](#)
- [MinIO Python SDK API文档](#)

贡献

[贡献指南](#)

Python Client API 文档

初始化 **MinIO Client** 对象。

MinIO

```
from minio import Minio
from minio.error import ResponseError

minioClient = Minio('play.min.io',
                    access_key='Q3AM3UQ867SPQQA43P2F',
                    secret_key='zuf+tfteSlsRu7BJ86wekitnifILbZam1KYY3TG',
                    secure=True)
```

AWS S3

```
from minio import Minio
from minio.error import ResponseError

s3Client = Minio('s3.amazonaws.com',
                  access_key='YOUR-ACCESSKEYID',
                  secret_key='YOUR-SECRETACCESSKEY',
                  secure=True)
```

操作存储桶	操作对象	Presigned操作	存储桶策略/通知
make_bucket	get_object	presigned_get_object	get_bucket_policy
list_buckets	put_object	presigned_put_object	set_bucket_policy
bucket_exists	_object	presigned_post_policy	get_bucket_notification
remove_bucket	stat_object		set_bucket_notification
list_objects	remove_object		remove_all_bucket_notification
list_objects_v2	remove_objects		listen_bucket_notification
list_incomplete_uploads	remove_incomplete_upload		
	fput_object		
	fget_object		
	get_partial_object		

1. 构造函数

Minio(endpoint, access_key=None, secret_key=None, secure=True, region=None, http_client=None)

```
Minio(endpoint, access_key=None, secret_key=None, secure=True, region=None, http_client=None)
```

初始化一个新的client对象。

参数

参数	类型	描述

<code>endpoint</code>	<code>string</code>	S3兼容对象存储服务endpoint。
<code>access_key</code>	<code>string</code>	对象存储的Access key。（如果是匿名访问则可以为空）。
<code>secret_key</code>	<code>string</code>	对象存储的Secret key。（如果是匿名访问则可以为空）。
<code>secure</code>	<code>bool</code>	设为 <code>True</code> 代表启用HTTPS。（默认是 <code>True</code> ）。
<code>region</code>	<code>string</code>	设置该值以覆盖自动发现存储桶region。（可选，默认值是 <code>None</code> ）。
<code>http_client</code>	<code>urllib3.PoolManager</code>	设置该值以使用自定义的http client，而不是默认的http client。（可选，默认值是 <code>None</code> ）。

示例

MinIO

```
from minio import Minio
from minio.error import ResponseError

minioClient = Minio('play.min.io',
                    access_key='Q3AM3UQ867SPQQA43P2F',
                    secret_key='zuf+tfeS1swRu7BJ86wekitnifILbZam1KYY3TG')
from minio import Minio
from minio.error import ResponseError
import urllib3

httpClient = urllib3.ProxyManager(
    'https://proxy_host.sampledomain.com:8119/',
    timeout=urllib3.Timeout.DEFAULT_TIMEOUT,
    cert_reqs='CERT_REQUIRED',
    retries=urllib3.Retry(
        total=5,
        backoff_factor=0.2,
        status_forcelist=[500, 502, 503, 504]
    )
)
minioClient = Minio('your_hostname.sampledomain.com:9000',
                    access_key='ACCESS_KEY',
                    secret_key='SECRET_KEY',
                    secure=True,
                    http_client=httpClient)
```

AWS S3

```
from minio import Minio
from minio.error import ResponseError

s3Client = Minio('s3.amazonaws.com',
                  access_key='ACCESS_KEY',
                  secret_key='SECRET_KEY')
```

2. 操作存储桶

`make_bucket(bucket_name, location='us-east-1')`

创建一个存储桶。

参数

参数	类型	描述
bucket_name	string	存储桶名称。
location	string	存储桶被创建的region(地区), 默认是us-east-1(美国东一区), 下面列举的是其它合法的值:
		us-east-1
		us-west-1
		us-west-2
		eu-west-1
		eu-central-1
		ap-southeast-1
		ap-northeast-1
		ap-southeast-2
		sa-east-1
		cn-north-1

示例

```
try:  
    minioClient.make_bucket("mybucket", location="us-east-1")  
except ResponseError as err:  
    print(err)
```

list_buckets()

列出所有的存储桶。

参数

返回值	类型	描述
bucketList	function	所有存储桶的list。
bucket.name	string	存储桶名称。
bucket.creation_date	time	存储桶的创建时间。

示例

```
buckets = minioClient.list_buckets()  
for bucket in buckets:  
    print(bucket.name, bucket.creation_date)
```

bucket_exists(bucket_name)

检查存储桶是否存在。

参数

参数	类型	描述
bucket_name	string	存储桶名称。

示例

```

try:
    print(minioClient.bucket_exists("mybucket"))
except ResponseError as err:
    print(err)

```

remove_bucket(bucket_name)

删除存储桶。

参数

参数	类型	描述
bucket_name	string	存储桶名称。

示例

```

try:
    minioClient.remove_bucket("mybucket")
except ResponseError as err:
    print(err)

```

list_objects(bucket_name, prefix=None, recursive=False)

列出存储桶中所有对象。

参数

参数	类型	描述
bucket_name	string	存储桶名称。
prefix	string	用于过滤的对象名称前缀。可选项， 默认为None。
recursive	bool	True 代表递归查找， False 代表类似文件夹查找，以'/'分隔，不查子文件夹。（可选， 默认值是 False ）。

返回值

参数	类型	描述
object	Object	该存储桶中所有对象的Iterator，对象的格式如下：

参数	类型	描述
object.bucket_name	string	对象所在存储桶的名称。
object.object_name	string	对象的名称。
object.is_dir	bool	True 代表列举的对象是文件夹（对象前缀）， False 与之相反。
object.size	int	对象的大小。
object.etag	string	对象的etag值。
object.last_modified	datetime.datetime	最后修改时间。
object.content_type	string	对象的content-type。
object.metadata	dict	对象的其它元数据。

示例

```
# List all object paths in bucket that begin with my-prefixname.
```

```

objects = minioClient.list_objects('mybucket', prefix='my-prefixname',
                                   recursive=True)
for obj in objects:
    print(obj.bucket_name, obj.object_name.encode('utf-8'), obj.last_modified,
          obj.etag, obj.size, obj.content_type)

```

list_objects_v2(bucket_name, prefix=None, recursive=False)

使用V2版本API列出一个存储桶中的对象。

参数

参数	类型	描述
bucket_name	string	存储桶名称。
prefix	string	用于过滤的对象名称前缀。可选项，默认为None。
recursive	bool	True 代表递归查找， False 代表类似文件夹查找，以'/'分隔，不查子文件夹。（可选， 默认值是 False ）。

返回值

参数	类型	描述
object	Object	该存储桶中所有对象的Iterator，对象的格式如下：

参数	类型	描述
object.bucket_name	string	对象所在存储桶的名称。
object.object_name	string	对象的名称。
object.is_dir	bool	True 代表列举的对象是文件夹（对象前缀）， False 与之相反。
object.size	int	对象的大小。
object.etag	string	对象的etag值。
object.last_modified	datetime.datetime	最后修改时间。
object.content_type	string	对象的content-type。
object.metadata	dict	对象的其它元数据。

示例

```

# List all object paths in bucket that begin with my-prefixname.
objects = minioClient.list_objects_v2('mybucket', prefix='my-prefixname',
                                       recursive=True)
for obj in objects:
    print(obj.bucket_name, obj.object_name.encode('utf-8'), obj.last_modified,
          obj.etag, obj.size, obj.content_type)

```

list_incomplete_uploads(bucket_name, prefix, recursive=False)

列出存储桶中未完整上传的对象。

参数

参数	类型	描述
bucket_name	string	存储桶名称。
prefix	string	用于过滤的对象名称前缀。

<code>recursive</code>	<code>bool</code>	<code>True</code> 代表递归查找, <code>False</code> 代表类似文件夹查找, 以'/'分隔, 不查子文件夹。 (可选, 默认值是 <code>False</code>)。
------------------------	-------------------	---

返回值

参数	类型	描述
<code>multipart_obj</code>	<code>Object</code>	<code>multipart</code> 对象的Iterator, 格式如下:

参数	类型	描述
<code>multipart_obj.object_name</code>	<code>string</code>	未完整上传的对象的名称。
<code>multipart_obj.upload_id</code>	<code>string</code>	未完整上传的对象的上传ID。
<code>multipart_obj.size</code>	<code>int</code>	未完整上传的对象的大小。

示例

```
# List all object paths in bucket that begin with my-prefixname.
uploads = minioClient.list_incomplete_uploads('mybucket',
                                                prefix='my-prefixname',
                                                recursive=True)

for obj in uploads:
    print(obj.bucket_name, obj.object_name, obj.upload_id, obj.size)
```

get_bucket_policy(bucket_name, prefix)

获取存储桶的当前策略。

参数

参数	类型	描述
<code>bucket_name</code>	<code>string</code>	存储桶名称。
<code>prefix</code>	<code>string</code>	对象的名称前缀。

返回值

参数	类型	描述
<code>Policy</code>	<code>minio.policy.Policy</code>	Policy枚举: Policy.READ_ONLY, Policy.WRITE_ONLY, Policy.READ_WRITE 或 Policy.NONE。

示例

```
# Get current policy of all object paths in bucket that begin with my-prefixname.
policy = minioClient.get_bucket_policy('mybucket',
                                       'my-prefixname')
print(policy)
```

set_bucket_policy(bucket_name, prefix, policy)

给指定的存储桶设置存储桶策略。如果 `prefix` 不为空, 则该存储桶策略仅对匹配这个指定前缀的对象生效。

参数

参数	类型	描述
<code>bucket_name</code>	<code>string</code>	存储桶名称。
<code>prefix</code>	<code>string</code>	对象的名称前缀。

Policy

minio.policy.Policy

Policy枚举: Policy.READ_ONLY, Policy.WRITE_ONLY, Policy.READ_WRITE或 Policy.NONE。

示例

```
# Set policy Policy.READ_ONLY to all object paths in bucket that begin with my-prefixname.
minioClient.set_bucket_policy('mybucket',
                             'my-prefixname',
                             Policy.READ_ONLY)
```

get_bucket_notification(bucket_name)

获取存储桶上的通知配置。

参数

参数	类型	描述
bucket_name	<i>string</i>	存储桶名称。

返回值

参数	类型	描述
notification	<i>dict</i>	如果没有通知配置，则返回一个空的 <i>dictionary</i> , 否则就和set_bucket_notification的参数结构一样。

示例

```
# Get the notifications configuration for a bucket.
notification = minioClient.get_bucket_notification('mybucket')
# If no notification is present on the bucket:
# notification == {}
```

set_bucket_notification(bucket_name, notification)

给存储桶设置通知配置。

参数

参数	类型	描述
bucket_name	<i>string</i>	存储桶名称。
notification	<i>dict</i>	非空 <i>dictionary</i> , 内部结构格式如下:

notification 参数格式如下:

- (*dict*) --
 - **TopicConfigurations** (*list*) -- 服务配置项目的可选列表，指定了AWS SNS Topics做为通知的目标。
 - **QueueConfigurations** (*list*) -- 服务配置项目的可选列表，指定了AWS SQS Queues做为通知的目标。
 - **CloudFunctionconfigurations** (*list*) -- 服务配置项目的可选列表，指定了AWS Lambda Cloud functions做为通知的目标。

以上项目中至少有一项需要在 `notification` 参数中指定。

上面提到的“服务配置项目”具有以下结构:

- (*dict*) --
 - **Id** (*string*) -- 配置项的可选ID, 如果不指定, 服务器自动生成。

- **Arn** (string) -- 指定特定的Topic/Queue/Cloud Function identifier。
- **Events** (list) -- 一个含有事件类型字符串的非空列表，事件类型取值如下： 's3:ReducedRedundancyLostObject', 's3:ObjectCreated:', 's3:ObjectCreated:Put', 's3:ObjectCreated:Post', 's3:ObjectCreated:', 's3:ObjectCreated:CompleteMultipartUpload', 's3:ObjectRemoved:*', 's3:ObjectRemoved:Delete', 's3:ObjectRemoved:DeleteMarkerCreated'*
- **Filter** (dict) -- 一个可选的dictionary容器，里面含有基于键名称过滤的规则的对象。
- **Key**
 - (dict) -- dictionary容器，里面含有基于键名称前缀和后缀过滤的规则的对象。
 - **FilterRules** (list) -- 指定过滤规则标准的容器列表。
 - (dict) -- 键值对的dictionary容器，指定单个的过滤规则。
 - **Name** (string) -- 对象的键名称，值为“前缀”或“后缀”。
 - **Value** (string) -- 指定规则适用的值。

没有返回值。如果目标服务报错，会抛出 `ResponseError`。如果有验证错误，会抛出 `InvalidArgumentException` 或者 `TypeError`。输入参数的`configuration`不能为空 - 为了删除存储桶上的通知配置，参考 `remove_all_bucket_notification()` API。

示例

```
notification = {
    'QueueConfigurations': [
        {
            'Id': '1',
            'Arn': 'arn1',
            'Events': ['s3:ObjectCreated:*'],
            'Filter': {
                'Key': {
                    'FilterRules': [
                        {
                            'Name': 'prefix',
                            'Value': 'abc'
                        }
                    ]
                }
            }
        },
        ],
    'TopicConfigurations': [
        {
            'Arn': 'arn2',
            'Events': ['s3:ObjectCreated:*'],
            'Filter': {
                'Key': {
                    'FilterRules': [
                        {
                            'Name': 'suffix',
                            'Value': '.jpg'
                        }
                    ]
                }
            }
        },
        ],
    'CloudFunctionConfigurations': [
        {
            'Arn': 'arn3',

```

```

'Events': ['s3:ObjectRemoved:*'],
'Filter': {
    'Key': {
        'FilterRules': [
            {
                'Name': 'suffix',
                'Value': '.jpg'
            }
        ]
    }
}
}

try:
    minioClient.set_bucket_notification('mybucket', notification)
except ResponseError as err:
    # handle error response from service.
    print(err)
except (ArgumentError, TypeError) as err:
    # should happen only during development. Fix the notification argument
    print(err)

```

remove_all_bucket_notification(bucket_name)

删除存储桶上配置的所有通知。

参数

参数	类型	描述
bucket_name	string	存储桶名称。

没有返回值，如果操作失败会抛出 `ResponseError` 异常。

示例

```

# Remove all the notifications config for a bucket.
minioClient.remove_all_bucket_notification('mybucket')

```

listen_bucket_notification(bucket_name, prefix, suffix, events)

监听存储桶上的通知，可以额外提供前缀、后缀和时间类型来进行过滤。使用该API前不需要先设置存储桶通知。这是一个MinIO的扩展API，MinIO Server会基于过来的请求使用唯一标识符自动注册或者注销。

当通知发生时，产生事件，调用者需要遍历读取这些事件。

参数

参数	类型	描述
bucket_name	string	监听事件通知的存储桶名称。
prefix	string	过滤通知的对象名称前缀。
suffix	string	过滤通知的对象名称后缀。
events	list	启用特定事件类型的通知。

完整示例请看 [这里](#)。

```

# Put a file with default content-type.
events = minioClient.listen_bucket_notification('my-bucket', 'my-prefix/',
                                                '.my-suffix',
                                                ['s3:ObjectCreated:*',
                                                 's3:ObjectRemoved:*',
                                                 's3:ObjectAccessed:*'])

for event in events:
    print event

```

3. 操作对象

get_object(bucket_name, object_name, request_headers=None)

下载一个对象。

参数

参数	类型	描述
bucket_name	string	存储桶名称。
object_name	string	对象名称。
request_headers	dict	额外的请求头信息（可选，默认为None）。

返回值

参数	类型	描述
object	urllib3.response.HTTPResponse	http streaming reader。

示例

```

# Get a full object.
try:
    data = minioClient.get_object('mybucket', 'myobject')
    with open('my-testfile', 'wb') as file_data:
        for d in data.stream(32*1024):
            file_data.write(d)
except ResponseError as err:
    print(err)

```

get_partial_object(bucket_name, object_name, offset=0, length=0, request_headers=None)

下载一个对象的指定区间的字节数组。

参数

参数	类型	描述		
bucket_name	string	存储桶名称。		
object_name	string	对象名称。		
offset	int	offset 是起始字节的位置		
length	int	length 是要读取的长度（可选，如果无值则代表读到文件结尾）。		
request_headers	dict	额外的请求头信息（可选，默认为None）。		

返回值

参数	类型	描述
object	<code>urllib3.response.HTTPResponse</code>	http streaming reader。

示例

```
# Offset the download by 2 bytes and retrieve a total of 4 bytes.
try:
    data = minioClient.get_partial_object('mybucket', 'myobject', 2, 4)
    with open('my-testfile', 'wb') as file_data:
        for d in data:
            file_data.write(d)
except ResponseError as err:
    print(err)
```

fget_object(bucket_name, object_name, file_path, request_headers=None)

下载并将文件保存到本地。

参数

参数	类型	描述
bucket_name	<code>string</code>	存储桶名称。
object_name	<code>string</code>	对象名称。
file_path	<code>dict</code>	对象数据要写入的本地文件路径。
request_headers	<code>dict</code>	额外的请求头信息（可选，默认为None）。

返回值

参数	类型	描述
obj	<code>Object</code>	对象的统计信息，格式如下：

参数	类型	描述
obj.size	<code>int</code>	对象的大小。
obj.etag	<code>string</code>	对象的etag值。
obj.content_type	<code>string</code>	对象的Content-Type。
obj.last_modified	<code>time.time</code>	最后修改时间。
obj.metadata	<code>dict</code>	对象的其它元数据。

示例

```
# Get a full object and prints the original object stat information.
try:
    print(minioClient.fget_object('mybucket', 'myobject', '/tmp/myobject'))
except ResponseError as err:
    print(err)
```

_object(bucket_name, object_name, object_source, _conditions=None, metadata=None)

拷贝对象存储服务上的源对象到一个新对象。

注意：本API支持的最大文件大小是5GB。

参数

参数	类型	描述
bucket_name	string	新对象的存储桶名称。
object_name	string	新对象的名称。
object_source	string	要拷贝的源对象的存储桶名称+对象名称。
_conditions	Conditions	拷贝操作需要满足的一些条件（可选， 默认为None）。

示例

以下所有条件都是允许的，并且可以组合使用。

```
import time
from datetime import datetime
from minio import Conditions

_conditions = Conditions()
# Set modified condition, object modified since 2014 April.
t = (2014, 4, 0, 0, 0, 0, 0, 0, 0)
mod_since = datetime.utcnow().timestamp()
_conditions.set_modified_since(mod_since)

# Set unmodified condition, object unmodified since 2014 April.
_conditions.set_unmodified_since(mod_since)

# Set matching ETag condition, object which matches the following ETag.
_conditions.set_match_etag("31624deb84149d2f8ef9c385918b653a")

# Set matching ETag except condition, object which does not match the following ETag.
_conditions.set_match_etag_except("31624deb84149d2f8ef9c385918b653a")

# Set metadata
metadata = {"test-key": "test-data"}

try:
    _result = minioClient._object("mybucket", "myobject",
                                  "/my-sourcebucketname/my-sourceobjectname",
                                  _conditions, metadata=metadata)
    print(_result)
except ResponseError as err:
    print(err)
```

put_object(bucket_name, object_name, data, length, content_type='application/octet-stream', metadata=None)

添加一个新的对象到对象存储服务。

注意：本API支持的最大文件大小是5TB。

参数

参数	类型	描述
bucket_name	string	存储桶名称。
object_name	string	对象名称。
data	io.RawIOBase	任何实现了io.RawIOBase的python对象。
length	int	对象的总长度。

<code>content_type</code>	<code>string</code>	对象的Content type。（可选， 默认是“application/octet-stream”）。
<code>metadata</code>	<code>dict</code>	其它元数据。（可选， 默认是None）。

返回值

参数	类型	描述
<code>etag</code>	<code>string</code>	对象的etag值。

示例

单个对象的最大大小限制在5TB。`put_object`在对象大于5MiB时，自动使用multiple parts方式上传。这样，当上传失败时，客户端只需要上传未成功的部分即可（类似断点上传）。上传的对象使用MD5SUM签名进行完整性验证。

```
import os
# Put a file with default content-type, upon success prints the etag identifier computed by server.
try:
    with open('my-testfile', 'rb') as file_data:
        file_stat = os.stat('my-testfile')
        print(minioClient.put_object('mybucket', 'myobject',
                                     file_data, file_stat.st_size))

except ResponseError as err:
    print(err)

# Put a file with 'application/csv'.
try:
    with open('my-testfile.csv', 'rb') as file_data:
        file_stat = os.stat('my-testfile.csv')
        minioClient.put_object('mybucket', 'myobject.csv', file_data,
                               file_stat.st_size, content_type='application/csv')

except ResponseError as err:
    print(err)
```

fput_object(bucket_name, object_name, file_path, content_type='application/octet-stream', metadata=None)

通过文件上传到对象中。

参数

参数	类型	描述
<code>bucket_name</code>	<code>string</code>	存储桶名称。
<code>object_name</code>	<code>string</code>	对象名称。
<code>file_path</code>	<code>string</code>	本地文件的路径，会将该文件的内容上传到对象存储服务上。
<code>content_type</code>	<code>string</code>	对象的Content type（可选， 默认是“application/octet-stream”）。
<code>metadata</code>	<code>dict</code>	其它元数据（可选， 默认是None）。

返回值

参数	类型	描述
<code>etag</code>	<code>string</code>	对象的etag值。

示例

单个对象的最大大小限制在5TB。`fput_object`在对象大于5MiB时，自动使用multiple parts方式上传。这样，当上传失败时，客户端只需要上传未成功的部分即可（类似断点上传）。上传的对象使用MD5SUM签名进行完整性验证。

```

# Put an object 'myobject' with contents from '/tmp/otherobject', upon success prints the etag identifier computed by server.
try:
    print(minioClient.fput_object('mybucket', 'myobject', '/tmp/otherobject'))
except ResponseError as err:
    print(err)

# Put on object 'myobject.csv' with contents from
# '/tmp/otherobject.csv' as 'application/csv'.
try:
    print(minioClient.fput_object('mybucket', 'myobject.csv',
                                  '/tmp/otherobject.csv',
                                  content_type='application/csv'))
except ResponseError as err:
    print(err)

```

stat_object(bucket_name, object_name)

获取对象的元数据。

参数

参数	类型	描述
bucket_name	string	存储桶名称。
object_name	string	名称名称。

返回值

参数	类型	描述
obj	Object	对象的统计信息，格式如下：

参数	类型	描述
obj.size	int	对象的大小。
obj.etag	string	对象的etag值。
obj.content_type	string	对象的Content-Type。
obj.last_modified	time.time	UTC格式的最后修改时间。
obj.metadata	dict	对象的其它元数据信息。

示例

```

# Fetch stats on your object.
try:
    print(minioClient.stat_object('mybucket', 'myobject'))
except ResponseError as err:
    print(err)

```

remove_object(bucket_name, object_name)

删除一个对象。

参数

参数	类型	描述
bucket_name	string	存储桶名称。

<code>object_name</code>	<code>string</code>	对象名称。
--------------------------	---------------------	-------

示例

```
# Remove an object.
try:
    minioClient.remove_object('mybucket', 'myobject')
except ResponseError as err:
    print(err)
```

remove_objects(bucket_name, objects_iter)

删除存储桶中的多个对象。

参数

参数	类型	描述
<code>bucket_name</code>	<code>string</code>	存储桶名称。
<code>objects_iter</code>	<code>list, tuple or iterator</code>	多个对象名称的列表数据。

返回值

参数	类型	描述
<code>delete_error_iterator</code>	<code>iterator of MultiDeleteError instances</code>	删除失败的错误信息iterator,格式如下:

注意

1. 由于上面的方法是延迟计算（lazy evaluation），默认是不计算的，所以上面返回的iterator必须被evaluated（比如：使用循环）。
2. 该iterator只有在执行删除操作出现错误时才不为空，每一项都包含删除报错的对象的错误信息。

该iterator产生的每一个删除错误信息都有如下结构：

参数	类型	描述
<code>MultiDeleteError.object_name</code>	<code>string</code>	删除报错的对象名称。
<code>MultiDeleteError.error_code</code>	<code>string</code>	错误码。
<code>MultiDeleteError.error_message</code>	<code>string</code>	错误信息。

示例

```
# Remove multiple objects in a single library call.
try:
    objects_to_delete = ['myobject-1', 'myobject-2', 'myobject-3']
    # force evaluation of the remove_objects() call by iterating over
    # the returned value.
    for del_err in minioClient.remove_objects('mybucket', objects_to_delete):
        print("Deletion Error: {}".format(del_err))
except ResponseError as err:
    print(err)
```

remove_incomplete_upload(bucket_name, object_name)

删除一个未完整上传的对象。

参数

参数	类型	描述
bucket_name	string	存储桶名称。
object_name	string	对象名称。

示例

```
# Remove an partially uploaded object.
try:
    minioClient.remove_incomplete_upload('mybucket', 'myobject')
except ResponseError as err:
    print(err)
```

4. Presigned操作

presigned_get_object(bucket_name, object_name, expiry=timedelta(days=7))

生成一个用于HTTP GET操作的presigned URL。浏览器/移动客户端可以在即使存储桶为私有的情况下也可以通过这个URL进行下载。这个presigned URL可以有一个过期时间，默认是7天。

参数

参数	类型	描述
bucket_name	string	存储桶名称。
object_name	string	对象名称。
expiry	datetime.datetime	过期时间，单位是秒，默认是7天。
response_headers	dictionary	额外的响应头（比如： response-content-type 、 response-content-disposition ）。

示例

```
from datetime import timedelta

# presigned get object URL for object name, expires in 2 days.
try:
    print(minioClient.presigned_get_object('mybucket', 'myobject', expires=timedelta(days=2)))
# Response error is still possible since internally presigned does get bucket location.
except ResponseError as err:
    print(err)
```

presigned_put_object(bucket_name, object_name, expires=timedelta(days=7))

生成一个用于HTTP PUT操作的presigned URL。浏览器/移动客户端可以在即使存储桶为私有的情况下也可以通过这个URL进行上传。这个presigned URL可以有一个过期时间，默认是7天。

注意：你可以通过只指定对象名称上传到S3。

参数

参数	类型	描述
bucket_name	string	存储桶名称。
object_name	string	对象名称。
expiry	datetime.datetime	过期时间，单位是秒，默认是7天。

示例

```

from datetime import timedelta

# presigned Put object URL for an object name, expires in 3 days.
try:
    print(minioClient.presigned_put_object('mybucket',
                                           'myobject',
                                           expires=timedelta(days=3)))
# Response error is still possible since internally presigned does get
# bucket location.
except ResponseError as err:
    print(err)

```

presigned_post_policy(PostPolicy)

允许给POST操作的presigned URL设置策略条件。这些策略包括比如，接收对象上传的存储桶名称，名称前缀，过期策略。

创建policy:

```

from datetime import datetime, timedelta

from minio import PostPolicy
post_policy = PostPolicy()

# Apply upload policy restrictions:

# set bucket name location for uploads.
post_policy.set_bucket_name('mybucket')
# set key prefix for all incoming uploads.
post_policy.set_key_startswith('myobject')
# set content length for incoming uploads.
post_policy.set_content_length_range(10, 1024)
# set content-type to allow only text
post_policy.set_content_type('text/plain')

# set expiry 10 days into future.
expires_date = datetime.utcnow() + timedelta(days=10)
post_policy.set_expires(expires_date)

```

获得POST表单的键值对形式的对象:

```

try:
    signed_form_data = minioClient.presigned_post_policy(post_policy)
except ResponseError as err:
    print(err)

```

使用 curl POST你的数据:

```

curl_str = 'curl -X POST {0}'.format(signed_form_data[0])
curl_cmd = [curl_str]
for field in signed_form_data[1]:
    curl_cmd.append('-F {0}={1}'.format(field, signed_form_data[1][field]))

# print curl command to upload files.
curl_cmd.append('-F file=@<FILE>')
print(' '.join(curl_cmd))

```

5. 了解更多

- [MinIO Golang Client SDK快速入门](#)
- [MinIO Java Client SDK快速入门](#)
- [MinIO JavaScript Client SDK快速入门](#)

适用于与Amazon S3兼容云存储的MinIO Go SDK

MinIO Go Client SDK提供了简单的API来访问任何与Amazon S3兼容的对象存储服务。

支持的云存储：

- AWS Signature Version 4
 - Amazon S3
 - MinIO
- AWS Signature Version 2
 - Google Cloud Storage (兼容模式)
 - Openstack Swift + Swift3 middleware
 - Ceph Object Gateway
 - Riak CS

本文我们将学习如何安装MinIO client SDK，连接到MinIO，并提供一下文件上传的示例。对于完整的API以及示例，请参考[Go Client API Reference](#)。

本文假设你已经有[Go开发环境](#)。

从Github下载

```
go get -u github.com/minio/minio-go
```

初始化MinIO Client

MinIO client需要以下4个参数来连接与Amazon S3兼容的对象存储。

参数	描述
endpoint	对象存储服务的URL
accessKeyID	Access key是唯一标识你的账户的用户ID。
secretAccessKey	Secret key是你账户的密码。
secure	true代表使用HTTPS

```
package main

import (
    "github.com/minio/minio-go/v6"
    "log"
)

func main() {
    endpoint := "play.min.io"
    accessKeyID := "Q3AM3UQ867SPQQA43P2F"
    secretAccessKey := "zuf+tfteSlswRu7BJ86wekitnifILbZam1KYY3TG"
    useSSL := true

    // 初始化 minio client对象。
    minioClient, err := minio.New(endpoint, accessKeyID, secretAccessKey, useSSL)
    if err != nil {
        log.Fatalln(err)
    }
}
```

```
    log.Printf("%#v\n", minioClient) // minioClient初始化成功
}
```

示例-文件上传

本示例连接到一个对象存储服务，创建一个存储桶并上传一个文件到存储桶中。

我们在本示例中使用运行在 <https://play.min.io> 上的MinIO服务，你可以用这个服务来开发和测试。示例中的访问凭据是公开的。

FileUploader.go

```
package main

import (
    "github.com/minio/minio-go/v6"
    "log"
)

func main() {
    endpoint := "play.min.io"
    accessKeyID := "Q3AM3UQ867SPQQA43P2F"
    secretAccessKey := "zuf+tfeS1swRu7BJ86wekitnifILbZam1KYY3TG"
    useSSL := true

    // 初始化minio client对象。
    minioClient, err := minio.New(endpoint, accessKeyID, secretAccessKey, useSSL)
    if err != nil {
        log.Fatalln(err)
    }

    // 创建一个叫mymusic的存储桶。
    bucketName := "mymusic"
    location := "us-east-1"

    err = minioClient.MakeBucket(bucketName, location)
    if err != nil {
        // 检查存储桶是否已经存在。
        exists, err := minioClient.BucketExists(bucketName)
        if err == nil && exists {
            log.Printf("We already own %s\n", bucketName)
        } else {
            log.Fatalln(err)
        }
    }
    log.Printf("Successfully created %s\n", bucketName)

    // 上传一个zip文件。
    objectName := "golden-oldies.zip"
    filePath := "/tmp/golden-oldies.zip"
    contentType := "application/zip"

    // 使用FPutObject上传一个zip文件。
    n, err := minioClient.FPutObject(bucketName, objectName, filePath, minio.PutObjectOptions{ContentType:contentType})
    if err != nil {
        log.Fatalln(err)
    }
}
```

```
    log.Printf("Successfully uploaded %s of size %d\n", objectName, n)
}
```

运行FileUploader

```
go run file-uploader.go
2016/08/13 17:03:28 Successfully created mymusic
2016/08/13 17:03:40 Successfully uploaded golden-oldies.zip of size 16253413

mc ls play/mymusic/
[2016-05-27 16:02:16 PDT] 17MiB golden-oldies.zip
```

API文档

完整的API文档在这里。

- [完整API文档](#)

API文档 : 操作存储桶

- [MakeBucket](#)
- [ListBuckets](#)
- [BucketExists](#)
- [RemoveBucket](#)
- [ListObjects](#)
- [ListObjectsV2](#)
- [ListIncompleteUploads](#)

API文档 : 存储桶策略

- [SetBucketPolicy](#)
- [GetBucketPolicy](#)

API文档 : 存储桶通知

- [SetBucketNotification](#)
- [GetBucketNotification](#)
- [RemoveAllBucketNotification](#)
- [ListenBucketNotification](#) (MinIO Extension)

API文档 : 操作文件对象

- [FPutObject](#)
- [FGetObject](#)
- [FPutObjectWithContext](#)
- [FGetObjectWithContext](#)

API文档 : 操作对象

- [GetObject](#)
- [PutObject](#)
- [GetObjectWithContext](#)
- [PutObjectContext](#)

- [PutObjectStreaming](#)
- [StatObject](#)
- [Object](#)
- [RemoveObject](#)
- [RemoveObjects](#)
- [RemoveIncompleteUpload](#)

API文档: 操作加密对象

- [GetEncryptedObject](#)
- [PutEncryptedObject](#)

API文档 : Presigned操作

- [PresignedGetObject](#)
- [PresignedPutObject](#)
- [PresignedHeadObject](#)
- [PresignedPostPolicy](#)

API文档 : 客户端自定义设置

- [SetAppInfo](#)
- [SetCustomTransport](#)
- [TraceOn](#)
- [TraceOff](#)

完整示例

完整示例 : 操作存储桶

- [makebucket.go](#)
- [listbuckets.go](#)
- [bucketexists.go](#)
- [removebucket.go](#)
- [listobjects.go](#)
- [listobjectsV2.go](#)
- [listincompleteuploads.go](#)

完整示例 : 存储桶策略

- [setbucketpolicy.go](#)
- [getbucketpolicy.go](#)
- [listbucketpolicies.go](#)

完整示例 : 存储桶通知

- [setbucketnotification.go](#)
- [getbucketnotification.go](#)
- [removeallbucketnotification.go](#)
- [listenbucketnotification.go](#) (MinIO扩展)

完整示例 : 操作文件对象

- [fputobject.go](#)
- [fgetobject.go](#)
- [fputobject-context.go](#)
- [fgetobject-context.go](#)

完整示例：操作对象

- [putobject.go](#)
- [getobject.go](#)
- [putobject-context.go](#)
- [getobject-context.go](#)
- [statobject.go](#)
- [object.go](#)
- [removeobject.go](#)
- [removeincompleteupload.go](#)
- [removeobjects.go](#)

完整示例：操作加密对象

- [put-encrypted-object.go](#)
- [get-encrypted-object.go](#)
- [fput-encrypted-object.go](#)

完整示例：**Presigned**操作

- [presignedgetobject.go](#)
- [presignedputobject.go](#)
- [presignedheadobject.go](#)
- [presignedpostpolicy.go](#)

了解更多

- [完整文档](#)
- [MinIO Go Client SDK API文档](#)

贡献

[贡献指南](#)

MinIO Go Client API 文档

初始化 **MinIO Client** 对象。

MinIO

```
package main

import (
    "fmt"

    "github.com/minio/minio-go/v6"
)

func main() {
    // 使用ssl
    ssl := true

    // 初始化minio client对象。
    minioClient, err := minio.New("play.min.io", "Q3AM3UQ867SPQQA43P2F", "zuf+tfteSlsRu7BJ86wekitnifILbz
am1KYY3TG", ssl)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

AWS S3

```
package main

import (
    "fmt"

    "github.com/minio/minio-go/v6"
)

func main() {
    // 使用ssl
    ssl := true

    // 初始化minio client对象。
    s3Client, err := minio.New("s3.amazonaws.com", "YOUR-ACCESSKEYID", "YOUR-SECRETACCESSKEY", ssl)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

操作存储桶	操作对象	操作加密对象	Presigned操作	存储桶
MakeBucket	GetObject	NewSymmetricKey	PresignedGetObject	SetBucket
ListBuckets	PutObject	NewAsymmetricKey	PresignedPutObject	GetObject
BucketExists	Object	GetEncryptedObject	PresignedPostPolicy	SetBucket

RemoveBucket	StatObject	PutEncryptedObject		GetBucket
ListObjects	RemoveObject	NewSSEInfo		RemoveObject
ListObjectsV2	RemoveObjects	FPutEncryptedObject		ListObjectsV2
ListIncompleteUploads	RemoveIncompleteUpload			
	FPutObject			
	FGetObject			
	ComposeObject			
	NewSourceInfo			
	NewDestinationInfo			
	PutObjectWithContext			
	GetObjectWithContext			
	FPutObjectWithContext			
	FGetObjectWithContext			

1. 构造函数

New(endpoint, accessKeyID, secretAccessKey string, ssl bool) (*Client, error)

初始化一个新的client对象。

参数

参数	类型	描述
endpoint	string	S3兼容对象存储服务endpoint
accessKeyID	string	对象存储的Access key
secretAccessKey	string	对象存储的Secret key
ssl	bool	true代表使用HTTPS

NewWithRegion(endpoint, accessKeyID, secretAccessKey string, ssl bool, region string) (*Client, error)

初始化minio client,带有region配置。和New()不同的是，NewWithRegion避免了bucket-location操作，所以会快那么一丢丢。如果你的应用只使用一个region的话可以用这个方法。

参数

参数	类型	描述
endpoint	string	S3兼容对象存储服务endpoint
accessKeyID	string	对象存储的Access key
secretAccessKey	string	对象存储的Secret key
ssl	bool	true代表使用HTTPS
region	string	对象存储的region

2. 操作存储桶

MakeBucket(bucketName, location string) error

创建一个存储桶。

参数

参数	类型	描述
bucketName	string	存储桶名称
location	string	存储桶被创建的region(地区), 默认是us-east-1(美国东一区), 下面列举的是其它合法的值。注意: 如果用的是minio服务的话, region是在它的配置文件中, (默认是us-east-1)。
		us-east-1
		us-east-2
		us-west-1
		us-west-2
		ca-central-1
		eu-west-1
		eu-west-2
		eu-west-3
		eu-central-1
		eu-north-1
		ap-east-1
		ap-south-1
		ap-southeast-1
		ap-southeast-2
		ap-northeast-1
		ap-northeast-2
		ap-northeast-3
		me-south-1
		sa-east-1
		us-gov-west-1
		us-gov-east-1
		cn-north-1
		cn-northwest-1

示例

```
err = minioClient.MakeBucket("mybucket", "us-east-1")
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully created mybucket.")
```

ListBuckets() ([]BucketInfo, error)

列出所有的存储桶。

参数	类型	描述

bucketList

[]minio.BucketInfo

所有存储桶的list。

minio.BucketInfo

参数	类型	描述
bucket.Name	<i>string</i>	存储桶名称
bucket.CreationDate	<i>time.Time</i>	存储桶的创建时间

示例

```
buckets, err := minioClient.ListBuckets()
if err != nil {
    fmt.Println(err)
    return
}
for _, bucket := range buckets {
    fmt.Println(bucket)
}
```

BucketExists(bucketName string) (found bool, err error)

检查存储桶是否存在。

参数

参数	类型	描述
bucketName	<i>string</i>	存储桶名称

返回值

参数	类型	描述
found	<i>bool</i>	存储桶是否存在
err	<i>error</i>	标准Error

示例

```
found, err := minioClient.BucketExists("mybucket")
if err != nil {
    fmt.Println(err)
    return
}
if found {
    fmt.Println("Bucket found")
}
```

RemoveBucket(bucketName string) error

删除一个存储桶，存储桶必须为空才能被成功删除。

参数

参数	类型	描述
bucketName	<i>string</i>	存储桶名称

示例

```

err = minioClient.RemoveBucket("mybucket")
if err != nil {
    fmt.Println(err)
    return
}

```

ListObjects(bucketName, prefix string, recursive bool, doneCh chan struct{}) <-chan ObjectInfo

列举存储桶里的对象。

参数

参数	类型	描述
bucketName	string	存储桶名称
objectPrefix	string	要列举的对象前缀
recursive	bool	true 代表递归查找, false 代表类似文件夹查找, 以'/'分隔, 不查子文件夹。
doneCh	chan struct{}	在该channel上结束ListObjects iterator的一个message。

返回值

参数	类型	描述
objectInfo	chan minio.ObjectInfo	存储桶中所有对象的read channel, 对象的格式如下:

minio.ObjectInfo

属性	类型	描述
objectInfo.Key	string	对象的名称
objectInfo.Size	int64	对象的大小
objectInfo.ETag	string	对象的MD5校验码
objectInfo.LastModified	time.Time	对象的最后修改时间

```

// Create a done channel to control 'ListObjects' go routine.
doneCh := make(chan struct{})

// Indicate to our routine to exit cleanly upon return.
defer close(doneCh)

isRecursive := true
objectCh := minioClient.ListObjects("mybucket", "myprefix", isRecursive, doneCh)
for object := range objectCh {
    if object.Err != nil {
        fmt.Println(object.Err)
        return
    }
    fmt.Println(object)
}

```

ListObjectsV2(bucketName, prefix string, recursive bool, doneCh chan struct{}) <-chan ObjectInfo

使用listing API v2版本列举存储桶中的对象。

参数

参数	类型	描述
bucketName	string	存储桶名称
objectPrefix	string	要列举的对象前缀
recursive	bool	true 代表递归查找, false 代表类似文件夹查找, 以'/'分隔, 不查子文件夹。
doneCh	chan struct{}	在该channel上结束ListObjects iterator的一个message。

返回值

参数	类型	描述
objectInfo	chan minio.ObjectInfo	存储桶中所有对象的read channel

```
// Create a done channel to control 'ListObjectsV2' go routine.
doneCh := make(chan struct{})

// Indicate to our routine to exit cleanly upon return.
defer close(doneCh)

isRecursive := true
objectCh := minioClient.ListObjectsV2("mybucket", "myprefix", isRecursive, doneCh)
for object := range objectCh {
    if object.Err != nil {
        fmt.Println(object.Err)
        return
    }
    fmt.Println(object)
}
```

ListIncompleteUploads(bucketName, prefix string, recursive bool, doneCh chan struct{}) <- chan ObjectMultipartInfo

列举存储桶中未完整上传的对象。

参数

参数	类型	描述
bucketName	string	存储桶名称
prefix	string	未完整上传的对象的前缀
recursive	bool	true 代表递归查找, false 代表类似文件夹查找, 以'/'分隔, 不查子文件夹。
doneCh	chan struct{}	在该channel上结束ListIncompleteUploads iterator的一个message。

返回值

参数	类型	描述
multiPartInfo	chan minio.ObjectMultipartInfo	multipart对象格式如下:

minio.ObjectMultipartInfo

属性	类型	描述
multiPartObjInfo.Key	string	未完整上传的对象的名称
multiPartObjInfo.UploadID	string	未完整上传的对象的Upload ID
multiPartObjInfo.Size	int64	未完整上传的对象的大小

示例

```
// Create a done channel to control 'ListObjects' go routine.
doneCh := make(chan struct{})

// Indicate to our routine to exit cleanly upon return.
defer close(doneCh)

isRecursive := true // Recursively list everything at 'myprefix'
multiPartObjectCh := minioClient.ListIncompleteUploads("mybucket", "myprefix", isRecursive, doneCh)
for multiPartObject := range multiPartObjectCh {
    if multiPartObject.Err != nil {
        fmt.Println(multiPartObject.Err)
        return
    }
    fmt.Println(multiPartObject)
}
```

3. 操作对象

GetObject(bucketName, objectName string, opts GetObjectOptions) (*Object, error)

返回对象数据的流，`error`是读流时经常抛的那些错。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称
<code>objectName</code>	<code>string</code>	对象的名称
<code>opts</code>	<code>minio.GetObjectOptions</code>	GET请求的一些额外参数，像 <code>encryption</code> , <code>If-Match</code>

minio.GetObjectOptions

参数	类型	描述
<code>opts.Materials</code>	<code>encrypt.Materials</code>	<code>encrypt</code> 包提供的对流加密的接口，(更多信息，请看 https://godoc.org/github.com/minio/minio-go/v6)

返回值

参数	类型	描述
<code>object</code>	<code>*minio.Object</code>	<code>minio.Object</code> 代表了一个object reader。它实现了 <code>io.Reader</code> , <code>io.Seeker</code> , <code>io.ReaderAt</code> and <code>io.Closer</code> 接口。

示例

```
object, err := minioClient.GetObject("mybucket", "myobject", minio.GetObjectOptions{})
if err != nil {
    fmt.Println(err)
    return
}
localFile, err := os.Create("/tmp/local-file.jpg")
if err != nil {
    fmt.Println(err)
    return
}
if _, err = io.Copy(localFile, object); err != nil {
```

```

    fmt.Println(err)
    return
}

```

FGetObject(bucketName, objectName, filePath string, opts GetObjectOptions) error

下载并将文件保存到本地文件系统。

参数

参数	类型	描述
bucketName	string	存储桶名称
objectName	string	对象的名称
filePath	string	下载后保存的路径
opts	minio.GetObjectOptions	GET请求的一些额外参数, 像encryption, If-Match

示例

```

err = minioClient.FGetObject("mybucket", "myobject", "/tmp/myobject", minio.GetObjectOptions{})
if err != nil {
    fmt.Println(err)
    return
}

```

GetObjectWithContext(ctx context.Context, bucketName, objectName string, opts GetObjectOptions) (*Object, error)

和GetObject操作是一样的, 不过传入了取消请求的context。

参数

参数	类型	描述
ctx	context.Context	请求上下文 (Request context)
bucketName	string	存储桶名称
objectName	string	对象的名称
opts	minio.GetObjectOptions	GET请求的一些额外参数, 像encryption, If-Match

返回值

参数	类型	描述
object	*minio.Object	minio.Object代表了一个object reader。它实现了io.Reader, io.Seeker, io.ReaderAt and io.Closer接口。

示例

```

ctx, cancel := context.WithTimeout(context.Background(), 100 * time.Second)
defer cancel()

object, err := minioClient.GetObjectWithContext(ctx, "mybucket", "myobject", minio.GetObjectOptions{})
if err != nil {
    fmt.Println(err)
    return
}

```

```

localFile, err := os.Create("/tmp/local-file.jpg")
if err != nil {
    fmt.Println(err)
    return
}

if _, err = io.(localFile, object); err != nil {
    fmt.Println(err)
    return
}

```

FGetObjectWithContext(ctx context.Context, bucketName, objectName, filePath string, opts GetObjectOptions) error

和FGetObject操作是一样的，不过允许取消请求。

参数

参数	类型	描述
ctx	<i>context.Context</i>	请求上下文
bucketName	<i>string</i>	存储桶名称
objectName	<i>string</i>	对象的名称
filePath	<i>string</i>	下载后保存的路径
opts	<i>minio.GetObjectOptions</i>	GET请求的一些额外参数，像encryption, If-Match

示例

```

ctx, cancel := context.WithTimeout(context.Background(), 100 * time.Second)
defer cancel()

err = minioClient.FGetObjectWithContext(ctx, "mybucket", "myobject", "/tmp/myobject", minio.GetObjectOptions{
})
if err != nil {
    fmt.Println(err)
    return
}

```

FGetEncryptedObject(bucketName, objectName, filePath string, materials encrypt.Materials) error

和FGetObject操作是一样的，不过会对加密请求进行解密。

参数

参数	类型	描述
bucketName	<i>string</i>	存储桶名称
objectName	<i>string</i>	对象的名称
filePath	<i>string</i>	下载后保存的路径
materials	<i>encrypt.Materials</i>	encrypt 包提供的对流加密的接口，(更多信息，请看 https://godoc.org/github.com/minio/minio-go/v6)

示例

```

// Generate a master symmetric key
key := encrypt.NewSymmetricKey([]byte("my-secret-key-00"))

// Build the CBC encryption material
cbcMaterials, err := encrypt.NewCBCSecureMaterials(key)
if err != nil {
    fmt.Println(err)
    return
}

err = minioClient.FGetEncryptedObject("mybucket", "myobject", "/tmp/myobject", cbcMaterials)
if err != nil {
    fmt.Println(err)
    return
}

```

PutObject(bucketName, objectName string, reader io.Reader, objectSize int64, opts PutObjectOptions) (n int, err error)

当对象小于128MiB时，直接在一次PUT请求里进行上传。当大于128MiB时，根据文件的实际大小，PutObject会自动地将对象进行拆分成128MiB一块或更大一些进行上传。对象的最大大小是5TB。

参数

参数	类型	描述
bucketName	<i>string</i>	存储桶名称
objectName	<i>string</i>	对象的名称
reader	<i>io.Reader</i>	任意实现了 <i>io.Reader</i> 的GO类型
objectSize	<i>int64</i>	上传的对象的大小，-1代表未知。
opts	<i>minio.PutObjectOptions</i>	允许用户设置可选的自定义元数据，内容标题，加密密钥和用于分段上传操作的线程数量。

minio.PutObjectOptions

属性	类型	描述
opts.UserMetadata	<i>map[string]string</i>	用户元数据的Map
opts.Progress	<i>io.Reader</i>	获取上传进度的Reader
opts.ContentType	<i>string</i>	对象的Content type，例如"application/text"
opts.ContentEncoding	<i>string</i>	对象的Content encoding，例如"gzip"
opts.ContentDisposition	<i>string</i>	对象的Content disposition, "inline"
opts.CacheControl	<i>string</i>	指定针对请求和响应的缓存机制，例如"max-age=600"
opts.EncryptMaterials	<i>encrypt.Materials</i>	encrypt 包提供的对流加密的接口，(更多信息，请看 https://godoc.org/github.com/minio/minio-go/v6)

示例

```

file, err := os.Open("my-testfile")
if err != nil {
    fmt.Println(err)
    return
}
defer file.Close()

```

```

fileStat, err := file.Stat()
if err != nil {
    fmt.Println(err)
    return
}

n, err := minioClient.PutObject("mybucket", "myobject", file, fileStat.Size(), minio.PutObjectOptions{Content
Type:"application/octet-stream"})
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully uploaded bytes: ", n)

```

API方法在minio-go SDK版本v3.0.3中提供的PutObjectWithSize, PutObjectWithMetadata, PutObjectStreaming和PutObjectWithProgress被替换为接受指向PutObjectOptions struct的指针的新的PutObject调用变体。

PutObjectWithContext(ctx context.Context, bucketName, objectName string, reader io.Reader, objectSize int64, opts PutObjectOptions) (n int, err error)

和PutObject是一样的，不过允许取消请求。

参数

参数	类型	描述
ctx	<i>context.Context</i>	请求上下文
bucketName	<i>string</i>	存储桶名称
objectName	<i>string</i>	对象的名称
reader	<i>io.Reader</i>	任何实现 <i>io.Reader</i> 的Go类型
objectSize	<i>int64</i>	上传的对象的大小，-1代表未知
opts	<i>minio.PutObjectOptions</i>	允许用户设置可选的自定义元数据， <i>content-type</i> , <i>content-encoding</i> , <i>content-disposition</i> 以及 <i>cache-control headers</i> , 传递加密模块以加密对象，并可选地设置 <i>multipart put</i> 操作的线程数量。

示例

```

ctx, cancel := context.WithTimeout(context.Background(), 10 * time.Second)
defer cancel()

file, err := os.Open("my-testfile")
if err != nil {
    fmt.Println(err)
    return
}
defer file.Close()

fileStat, err := file.Stat()
if err != nil {
    fmt.Println(err)
    return
}

n, err := minioClient.PutObjectWithContext(ctx, "my-bucketname", "my-objectname", file, fileStat.Size(), mini
o.PutObjectOptions{
    ContentType: "application/octet-stream",
})
if err != nil {

```

```

        fmt.Println(err)
        return
    }
    fmt.Println("Successfully uploaded bytes: ", n)

```

Object(dst DestinationInfo, src SourceInfo) error

通过在服务端对已存在的对象进行拷贝，实现新建或者替换对象。它支持有条件的拷贝，拷贝对象的一部分，以及在服务端的加解密。请查看 `SourceInfo` 和 `DestinationInfo` 两个类型来了解更多细节。

拷贝多个源文件到一个目标对象，请查看 `ComposeObject API`。

参数

参数	类型	描述
<code>dst</code>	<code>minio.DestinationInfo</code>	目标对象
<code>src</code>	<code>minio.SourceInfo</code>	源对象

示例

```

// Use-case 1: Simple object with no conditions.
// Source object
src := minio.NewSourceInfo("my-sourcebucketname", "my-sourceobjectname", nil)

// Destination object
dst, err := minio.NewDestinationInfo("my-bucketname", "my-objectname", nil, nil)
if err != nil {
    fmt.Println(err)
    return
}

// object call
err = minioClient.Object(dst, src)
if err != nil {
    fmt.Println(err)
    return
}

// Use-case 2:
// object with -conditions, and ing only part of the source object.
// 1. that matches a given ETag
// 2. and modified after 1st April 2014
// 3. but unmodified since 23rd April 2014
// 4. only first 1MiB of object.

// Source object
src := minio.NewSourceInfo("my-sourcebucketname", "my-sourceobjectname", nil)

// Set matching ETag condition, object which matches the following ETag.
src.SetMatchETagCond("31624deb84149d2f8ef9c385918b653a")

// Set modified condition, object modified since 2014 April 1.
src.SetModifiedSinceCond(time.Date(2014, time.April, 1, 0, 0, 0, 0, time.UTC))

// Set unmodified condition, object unmodified since 2014 April 23.
src.SetUnmodifiedSinceCond(time.Date(2014, time.April, 23, 0, 0, 0, 0, time.UTC))

// Set -range of only first 1MiB of file.
src.SetRange(0, 1024*1024-1)

```

```

// Destination object
dst, err := minio.NewDestinationInfo("my-bucketname", "my-objectname", nil, nil)
if err != nil {
    fmt.Println(err)
    return
}

// object call
err = minioClient.Object(dst, src)
if err != nil {
    fmt.Println(err)
    return
}

```

ComposeObject(dst minio.DestinationInfo, srcs []minio.SourceInfo) error

通过使用服务端拷贝实现将多个源对象合并创建一个新的对象。

参数

参数	类型	描述
dst	<code>minio.DestinationInfo</code>	要被创建的目标对象
srcs	<code>[]minio.SourceInfo</code>	要合并的多个源对象

示例

```

// Prepare source decryption key (here we assume same key to
// decrypt all source objects.)
decKey := minio.NewSSEInfo([]byte{1, 2, 3}, "")

// Source objects to concatenate. We also specify decryption
// key for each
src1 := minio.NewSourceInfo("bucket1", "object1", &decKey)
src1.SetMatchETagCond("31624deb84149d2f8ef9c385918b653a")

src2 := minio.NewSourceInfo("bucket2", "object2", &decKey)
src2.SetMatchETagCond("f8ef9c385918b653a31624deb84149d2")

src3 := minio.NewSourceInfo("bucket3", "object3", &decKey)
src3.SetMatchETagCond("5918b653a31624deb84149d2f8ef9c38")

// Create slice of sources.
srcs := []minio.SourceInfo{src1, src2, src3}

// Prepare destination encryption key
encKey := minio.NewSSEInfo([]byte{8, 9, 0}, "")

// Create destination info
dst, err := minio.NewDestinationInfo("bucket", "object", &encKey, nil)
if err != nil {
    fmt.Println(err)
    return
}

// Compose object call by concatenating multiple source files.
err = minioClient.ComposeObject(dst, srcs)
if err != nil {
    fmt.Println(err)
    return
}

```

```

}

fmt.Println("Composed object successfully.")

```

NewSourceInfo(bucket, object string, decryptSSEC *SSEInfo) SourceInfo

构建一个可用于服务端拷贝操作（像 `Object` 和 `ComposeObject`）的 `SourceInfo` 对象。该对象可用于给源对象设置拷贝条件。

参数

参数	类型	描述
<code>bucket</code>	<code>string</code>	源存储桶
<code>object</code>	<code>string</code>	源对象
<code>decryptSSEC</code>	<code>*minio.SSEInfo</code>	源对象的解密信息 (<code>nil</code> 代表不用解密)

示例

```

// No decryption parameter.
src := minio.NewSourceInfo("bucket", "object", nil)

// Destination object
dst, err := minio.NewDestinationInfo("my-bucketname", "my-objectname", nil, nil)
if err != nil {
    fmt.Println(err)
    return
}

// object call
err = minioClient.Object(dst, src)
if err != nil {
    fmt.Println(err)
    return
}

// With decryption parameter.
deckey := minio.NewSSEInfo([]byte{1,2,3}, "")
src := minio.NewSourceInfo("bucket", "object", &deckey)

// Destination object
dst, err := minio.NewDestinationInfo("my-bucketname", "my-objectname", nil, nil)
if err != nil {
    fmt.Println(err)
    return
}

// object call
err = minioClient.Object(dst, src)
if err != nil {
    fmt.Println(err)
    return
}

```

NewDestinationInfo(bucket, object string, encryptSSEC *SSEInfo, userMeta map[string]string) (DestinationInfo, error)

构建一个用于服务端拷贝操作（像 `Object` 和 `ComposeObject`）的用作目标对象的 `DestinationInfo`。

参数

参数	类型	描述
bucket	<i>string</i>	目标存储桶名称
object	<i>string</i>	目标对象名称
encryptSSEC	<i>*minio.SSEInfo</i>	源对象的加密信息 (nil 代表不用加密)
userMeta	<i>map[string]string</i>	给目标对象的用户元数据，如果是nil，并只有一个源对象，则将源对象的用户元数据拷贝给目标对象。

示例

```
// No encryption parameter.  
src := minio.NewSourceInfo("bucket", "object", nil)  
dst, err := minio.NewDestinationInfo("bucket", "object", nil, nil)  
if err != nil {  
    fmt.Println(err)  
    return  
}  
  
// object call  
err = minioClient.Object(dst, src)  
if err != nil {  
    fmt.Println(err)  
    return  
}  
src := minio.NewSourceInfo("bucket", "object", nil)  
  
// With encryption parameter.  
encKey := minio.NewSSEInfo([]byte{1,2,3}, "")  
dst, err := minio.NewDestinationInfo("bucket", "object", &encKey, nil)  
if err != nil {  
    fmt.Println(err)  
    return  
}  
  
// object call  
err = minioClient.Object(dst, src)  
if err != nil {  
    fmt.Println(err)  
    return  
}
```

FPutObject(bucketName, objectName, filePath, opts PutObjectOptions) (length int64, err error)

将filePath对应的文件内容上传到一个对象中。

当对象小于128MiB时，FPutObject直接在一次PUT请求里进行上传。当大于128MiB时，根据文件的实际大小，FPutObject会自动地将对象进行拆分成128MiB一块或更大一些进行上传。对象的最大大小是5TB。

参数

参数	类型	描述
bucketName	<i>string</i>	存储桶名称
objectName	<i>string</i>	对象的名称
filePath	<i>string</i>	要上传的文件的路径

opts	<code>minio.PutObjectOptions</code>	允许用户设置可选的自定义元数据, <code>content-type</code> , <code>content-encoding</code> , <code>content-disposition</code> 以及 <code>cache-control headers</code> , 传递加密模块以加密对象, 并可选地设置multipart put操作的线程数量。
------	-------------------------------------	--

示例

```
n, err := minioClient.FPutObject("my-bucketname", "my-objectname", "my-filename.csv", minio.PutObjectOptions{
    ContentType: "application/csv",
})
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully uploaded bytes: ", n)
```

FPutObjectWithContext(ctx context.Context, bucketName, objectName, filePath, opts PutObjectOptions) (length int64, err error)

和FPutObject操作是一样的, 不过允许取消请求。

参数

参数	类型	描述
ctx	<code>context.Context</code>	请求上下文
bucketName	<code>string</code>	存储桶名称
objectName	<code>string</code>	对象的名称
filePath	<code>string</code>	要上传的文件的路径
opts	<code>minio.PutObjectOptions</code>	允许用户设置可选的自定义元数据, <code>content-type</code> , <code>content-encoding</code> , <code>content-disposition</code> 以及 <code>cache-control headers</code> , 传递加密模块以加密对象, 并可选地设置multipart put操作的线程数量。

示例

```
ctx, cancel := context.WithTimeout(context.Background(), 100 * time.Second)
defer cancel()

n, err := minioClient.FPutObjectWithContext(ctx, "mybucket", "myobject.csv", "/tmp/otherobject.csv", minio.PutObjectOptions{ContentType:"application/csv"})
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully uploaded bytes: ", n)
```

StatObject(bucketName, objectName string, opts StatObjectOptions) (ObjectInfo, error)

获取对象的元数据。

参数

参数	类型	描述
bucketName	<code>string</code>	存储桶名称
objectName	<code>string</code>	对象的名称
opts	<code>minio.StatObjectOptions</code>	GET info/stat请求的一些额外参数, 像 <code>encryption</code> , <code>If-Match</code>

返回值

参数	类型	描述
objInfo	minio.ObjectInfo	对象stat信息

minio.ObjectInfo

属性	类型	描述
objInfo.LastModified	time.Time	对象的最后修改时间
objInfo.ETag	string	对象的MD5校验码
objInfo.ContentType	string	对象的Content type
objInfo.Size	int64	对象的大小

示例

```
objInfo, err := minioClient.StatObject("mybucket", "myobject", minio.StatObjectOptions{})
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println(objInfo)
```

RemoveObject(bucketName, objectName string) error

删除一个对象。

参数

参数	类型	描述
bucketName	string	存储桶名称
objectName	string	对象的名称

```
err = minioClient.RemoveObject("mybucket", "myobject")
if err != nil {
    fmt.Println(err)
    return
}
```

RemoveObjects(bucketName string, objectsCh chan string) (errorCh <-chan RemoveObjectError)

从一个input channel里删除一个对象集合。一次发送到服务端的删除请求最多可删除1000个对象。通过error channel返回的错误信息。

参数

参数	类型	描述
bucketName	string	存储桶名称
objectsCh	chan string	要删除的对象的channel

返回值

参数	类型	描述

errorCh

<-chan minio.RemoveObjectError

删除时观察到的错误的Receive-only channel。

```
objectsCh := make(chan string)

// Send object names that are needed to be removed to objectsCh
go func() {
    defer close(objectsCh)
    // List all objects from a bucket-name with a matching prefix.
    for object := range minioClient.ListObjects("my-bucketname", "my-prefixname", true, nil) {
        if object.Err != nil {
            log.Fatalln(object.Err)
        }
        objectsCh <- object.Key
    }
}()

for rErr := range minioClient.RemoveObjects("mybucket", objectsCh) {
    fmt.Println("Error detected during deletion: ", rErr)
}
```

RemoveIncompleteUpload(bucketName, objectName string) error

删除一个未完整上传的对象。

参数

参数	类型	描述
bucketName	string	存储桶名称
objectName	string	对象的名称

示例

```
err = minioClient.RemoveIncompleteUpload("mybucket", "myobject")
if err != nil {
    fmt.Println(err)
    return
}
```

4. 操作加密对象

NewSymmetricKey(key []byte) *encrypt.SymmetricKey

参数

参数	类型	描述
key	string	存储桶名称

返回值

参数	类型	描述
symmetricKey	*encrypt.SymmetricKey	加密解密的对称秘钥

```
symKey := encrypt.NewSymmetricKey([]byte("my-secret-key-00"))

// Build the CBC encryption material with symmetric key.
```

```

cbcMaterials, err := encrypt.NewCBCSecureMaterials(symKey)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully initialized Symmetric key CBC materials", cbcMaterials)

object, err := minioClient.GetEncryptedObject("mybucket", "myobject", cbcMaterials)
if err != nil {
    fmt.Println(err)
    return
}
defer object.Close()

```

NewAsymmetricKey(privateKey []byte, publicKey[]byte) (*encrypt.AsymmetricKey, error)

参数

参数	类型	描述
privateKey	>[]byte	Private key数据
publicKey	>[]byte	Public key数据

返回值

参数	类型	描述
asymmetricKey	* <i>encrypt</i> .AsymmetricKey	加密解密的非对称秘钥
err	error	标准Error

```

privateKey, err := ioutil.ReadFile("private.key")
if err != nil {
    fmt.Println(err)
    return
}

publicKey, err := ioutil.ReadFile("public.key")
if err != nil {
    fmt.Println(err)
    return
}

// Initialize the asymmetric key
asymmetricKey, err := encrypt.NewAsymmetricKey(privateKey, publicKey)
if err != nil {
    fmt.Println(err)
    return
}

// Build the CBC encryption material for asymmetric key.
cbcMaterials, err := encrypt.NewCBCSecureMaterials(asymmetricKey)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully initialized Asymmetric key CBC materials", cbcMaterials)

object, err := minioClient.GetEncryptedObject("mybucket", "myobject", cbcMaterials)

```

```

if err != nil {
    fmt.Println(err)
    return
}
defer object.Close()

```

GetEncryptedObject(bucketName, objectName string, encryptMaterials encrypt.Materials) (io.ReadCloser, error)

返回对象的解密流。读流时的常见错误。

参数

参数	类型	描述
bucketName	string	存储桶名称
objectName	string	对象的名称
encryptMaterials	encrypt.Materials	encrypt 包提供的对流加密的接口, (更多信息, 请看 https://godoc.org/github.com/minio/minio-go/v6)

返回值

参数	类型	描述
stream	io.ReadCloser	返回对象的reader,调用者需要在读取之后进行关闭。
err	_error	错误信息

示例

```

// Generate a master symmetric key
key := encrypt.NewSymmetricKey([]byte("my-secret-key-00"))

// Build the CBC encryption material
cbcMaterials, err := encrypt.NewCBCSecureMaterials(key)
if err != nil {
    fmt.Println(err)
    return
}

object, err := minioClient.GetEncryptedObject("mybucket", "myobject", cbcMaterials)
if err != nil {
    fmt.Println(err)
    return
}
defer object.Close()

localFile, err := os.Create("/tmp/local-file.jpg")
if err != nil {
    fmt.Println(err)
    return
}
defer localFile.Close()

if _, err = io.(localFile, object); err != nil {
    fmt.Println(err)
    return
}

```

PutEncryptedObject(bucketName, objectName string, reader io.Reader, encryptMaterials encrypt.Materials) (n int, err error)

加密并上传对象。

参数

参数	类型	描述
bucketName	string	存储桶名称
objectName	string	对象的名称
reader	io.Reader	任何实现io.Reader的Go类型
encryptMaterials	encrypt.Materials	encrypt 包提供的对流加密的接口, (更多信息, 请看 https://godoc.org/github.com/minio/minio-go/v6)

示例

```
// Load a private key
privateKey, err := ioutil.ReadFile("private.key")
if err != nil {
    fmt.Println(err)
    return
}

// Load a public key
publicKey, err := ioutil.ReadFile("public.key")
if err != nil {
    fmt.Println(err)
    return
}

// Build an asymmetric key
key, err := encrypt.NewAsymmetricKey(privateKey, publicKey)
if err != nil {
    fmt.Println(err)
    return
}

// Build the CBC encryption module
cbcMaterials, err := encrypt.NewCBCSecureMaterials(key)
if err != nil {
    fmt.Println(err)
    return
}

// Open a file to upload
file, err := os.Open("my-testfile")
if err != nil {
    fmt.Println(err)
    return
}
defer file.Close()

// Upload the encrypted form of the file
n, err := minioClient.PutEncryptedObject("mybucket", "myobject", file, cbcMaterials)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully uploaded encrypted bytes: ", n)
```

FPutEncryptedObject(bucketName, objectName, filePath, encryptMaterials encrypt.Materials) (n int, err error)

通过一个文件进行加密并上传到对象。

参数

参数	类型	描述
bucketName	string	存储桶名称
objectName	string	对象的名称
filePath	string	要上传的文件的路径
encryptMaterials	encrypt.Materials	encrypt 包提供的对流加密的接口, (更多信息, 请看 https://godoc.org/github.com/minio/minio-go/v6)

示例

```
// Load a private key
privateKey, err := ioutil.ReadFile("private.key")
if err != nil {
    fmt.Println(err)
    return
}

// Load a public key
publicKey, err := ioutil.ReadFile("public.key")
if err != nil {
    fmt.Println(err)
    return
}

// Build an asymmetric key
key, err := encrypt.NewAsymmetricKey(privateKey, publicKey)
if err != nil {
    fmt.Println(err)
    return
}

// Build the CBC encryption module
cbcMaterials, err := encrypt.NewCBCSecureMaterials(key)
if err != nil {
    fmt.Println(err)
    return
}

n, err := minioClient.FPutEncryptedObject("mybucket", "myobject.csv", "/tmp/otherobject.csv", cbcMaterials)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully uploaded encrypted bytes: ", n)
```

NewSSEInfo(key []byte, algo string) SSEInfo

创建一个通过用户提供的key(SSE-C),进行服务端加解密操作的key对象。

参数

参数	类型	描述
key	<code>[]byte</code>	未编码的二进制key数组
algo	<code>string</code>	加密算法, 可以为空 (默认是 AES256)

5. Presigned操作

PresignedGetObject(bucketName, objectName string, expiry time.Duration, reqParams url.Values) (*url.URL, error)

生成一个用于HTTP GET操作的presigned URL。浏览器/移动客户端可以在即使存储桶为私有的情况下也可以通过这个URL进行下载。这个presigned URL可以有一个过期时间, 默认是7天。

参数

参数	类型	描述
bucketName	<code>string</code>	存储桶名称
objectName	<code>string</code>	对象的名称
expiry	<code>time.Duration</code>	presigned URL的过期时间, 单位是秒
reqParams	<code>url.Values</code>	额外的响应头, 支持 <code>response-expires</code> , <code>response-content-type</code> , <code>response-cache-control</code> , <code>response-content-disposition</code> 。

示例

```
// Set request parameters for content-disposition.
reqParams := make(url.Values)
reqParams.Set("response-content-disposition", "attachment; filename=\"your-filename.txt\"")

// Generates a presigned url which expires in a day.
presignedURL, err := minioClient.PresignedGetObject("mybucket", "myobject", time.Second * 24 * 60 * 60, reqParams)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully generated presigned URL", presignedURL)
```

PresignedPutObject(bucketName, objectName string, expiry time.Duration) (*url.URL, error)

生成一个用于HTTP PUT操作的presigned URL。浏览器/移动客户端可以在即使存储桶为私有的情况下也可以通过这个URL进行下载。这个presigned URL可以有一个过期时间, 默认是7天。

注意: 你可以通过只指定对象名称上传到S3。

参数

参数	类型	描述
bucketName	<code>string</code>	存储桶名称
objectName	<code>string</code>	对象的名称
expiry	<code>time.Duration</code>	presigned URL的过期时间, 单位是秒

示例

```

// Generates a url which expires in a day.
expiry := time.Second * 24 * 60 * 60 // 1 day.
presignedURL, err := minioClient.PresignedPutObject("mybucket", "myobject", expiry)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully generated presigned URL", presignedURL)

```

PresignedHeadObject(bucketName, objectName string, expiry time.Duration, reqParams url.Values) (*url.URL, error)

生成一个用于HTTP GET操作的presigned URL。浏览器/移动客户端可以在即使存储桶为私有的情况下也可以通过这个URL进行下载。这个presigned URL可以有一个过期时间，默认是7天。

参数

参数	类型	描述
bucketName	string	存储桶名称
objectName	string	对象的名称
expiry	time.Duration	presigned URL的过期时间，单位是秒
reqParams	url.Values	额外的响应头，支持response-expires, response-content-type, response-cache-control, response-content-disposition。

示例

```

// Set request parameters for content-disposition.
reqParams := make(url.Values)
reqParams.Set("response-content-disposition", "attachment; filename=\"your-filename.txt\"")

// Generates a presigned url which expires in a day.
presignedURL, err := minioClient.PresignedHeadObject("mybucket", "myobject", time.Second * 24 * 60 * 60, reqParams)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Successfully generated presigned URL", presignedURL)

```

PresignedPostPolicy(PostPolicy) (*url.URL, map[string]string, error)

允许给POST操作的presigned URL设置策略条件。这些策略包括比如，接收对象上传的存储桶名称，名称前缀，过期策略。

```

// Initialize policy condition config.
policy := minio.NewPostPolicy()

// Apply upload policy restrictions:
policy.SetBucket("mybucket")
policy.SetKey("myobject")
policy.SetExpires(time.Now().UTC().AddDate(0, 0, 10)) // expires in 10 days

// Only allow 'png' images.
policy.SetContentType("image/png")

```

```

// Only allow content size in range 1KB to 1MB.
policy.SetContentLengthRange(1024, 1024*1024)

// Add a user metadata using the key "custom" and value "user"
policy.SetUserMetadata("custom", "user")

// Get the POST form key/value object:
url, formData, err := minioClient.PresignedPostPolicy(policy)
if err != nil {
    fmt.Println(err)
    return
}

// POST your content from the command line using `curl`
fmt.Printf("curl ")
for k, v := range formData {
    fmt.Printf("-F %s=%s ", k, v)
}
fmt.Printf("-F file=@/etc/bash.bashrc ")
fmt.Printf("%s\n", url)

```

6. 存储桶策略/通知

SetBucketPolicy(bucketname, objectPrefix string, policy policy.BucketPolicy) error

给存储桶或者对象前缀设置访问权限。

必须引入 `github.com/minio/minio-go/v6/pkg/policy` 包。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称
<code>objectPrefix</code>	<code>string</code>	对象的名称前缀
<code>policy</code>	<code>policy.BucketPolicy</code>	Policy的取值如下:
		<code>policy.BucketPolicyNone</code>
		<code>policy.BucketPolicyReadOnly</code>
		<code>policy.BucketPolicyReadWrite</code>
		<code>policy.BucketPolicyWriteOnly</code>

返回值

参数	类型	描述
<code>err</code>	<code>error</code>	标准Error

示例

```

// Sets 'mybucket' with a sub-directory 'myprefix' to be anonymously accessible for
// both read and write operations.
err = minioClient.SetBucketPolicy("mybucket", "myprefix", policy.BucketPolicyReadWrite)
if err != nil {
    fmt.Println(err)
    return
}

```

GetBucketPolicy(bucketName, objectPrefix string) (policy.BucketPolicy, error)

获取存储桶或者对象前缀的访问权限。

必须引入 `github.com/minio/minio-go/v6/pkg/policy` 包。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称
<code>objectPrefix</code>	<code>string</code>	该存储桶下的对象前缀

返回值

参数	类型	描述
<code>bucketPolicy</code>	<code>policy.BucketPolicy</code>	取值如下: <code>none</code> , <code>readonly</code> , <code>readwrite</code> , 或者 <code>writeonly</code>
<code>err</code>	<code>error</code>	标准Error

示例

```
bucketPolicy, err := minioClient.GetBucketPolicy("mybucket", "")  
if err != nil {  
    fmt.Println(err)  
    return  
}  
fmt.Println("Access permissions for mybucket is", bucketPolicy)
```

GetBucketNotification(bucketName string) (BucketNotification, error)

获取存储桶的通知配置

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称

返回值

参数	类型	描述
<code>bucketNotification</code>	<code>minio.BucketNotification</code>	含有所有通知配置的数据结构
<code>err</code>	<code>error</code>	标准Error

示例

```
bucketNotification, err := minioClient.GetBucketNotification("mybucket")  
if err != nil {  
    fmt.Println("Failed to get bucket notification configurations for mybucket", err)  
    return  
}  
  
for _, queueConfig := range bucketNotification.QueueConfigs {  
    for _, e := range queueConfig.Events {  
        fmt.Println(e + " event is enabled")  
    }  
}
```

SetBucketNotification(bucketName string, bucketNotification BucketNotification) error

给存储桶设置新的通知

参数

参数	类型	描述
bucketName	string	存储桶名称
bucketNotification	minio.BucketNotification	发送给配置的web service的XML

返回值

参数	类型	描述
err	error	标准Error

示例

```
queueArn := minio.NewArn("aws", "sns", "us-east-1", "804605494417", "PhotoUpdate")

queueConfig := minio.NewNotificationConfig(queueArn)
queueConfig.AddEvents(minio.ObjectCreatedAll, minio.ObjectRemovedAll)
queueConfig.AddFilterPrefix("photos/")
queueConfig.AddFilterSuffix(".jpg")

bucketNotification := minio.BucketNotification{}
bucketNotification.AddQueue(queueConfig)

err = minioClient.SetBucketNotification("mybucket", bucketNotification)
if err != nil {
    fmt.Println("Unable to set the bucket notification: ", err)
    return
}
```

RemoveAllBucketNotification(bucketName string) error

删除存储桶上所有配置的通知

参数

参数	类型	描述
bucketName	string	存储桶名称

返回值

参数	类型	描述
err	error	标准Error

示例

```
err = minioClient.RemoveAllBucketNotification("mybucket")
if err != nil {
    fmt.Println("Unable to remove bucket notifications.", err)
    return
}
```

ListenBucketNotification(bucketName, prefix, suffix string, events []string, doneCh <-chan struct{}) <-chan NotificationInfo

ListenBucketNotification API通过notification channel接收存储桶通知事件。返回的notification channel有两个属性，'Records'和'Err'。

- 'Records'持有从服务器返回的通知信息。
- 'Err'表示的是处理接收到的通知时报的任何错误。

注意：一旦报错， notification channel就会关闭。

参数

参数	类型	描述
bucketName	string	被监听通知的存储桶
prefix	string	过滤通知的对象前缀
suffix	string	过滤通知的对象后缀
events	[]string	开启指定事件类型的通知
doneCh	chan struct{}	在该channel上结束ListenBucketNotification iterator的一个message。

返回值

参数	类型	描述
notificationInfo	chan minio.NotificationInfo	存储桶通知的channel

minio.NotificationInfo

|属性|类型|描述|| notificationInfo.Records | []minio.NotificationEvent | 通知事件的集合 || notificationInfo.Err | error | 操作时报的任何错误(标准Error) |

示例

```
// Create a done channel to control 'ListenBucketNotification' go routine.
doneCh := make(chan struct{})

// Indicate a background go-routine to exit cleanly upon return.
defer close(doneCh)

// Listen for bucket notifications on "mybucket" filtered by prefix, suffix and events.
for notificationInfo := range minioClient.ListenBucketNotification("mybucket", "myprefix/", ".mysuffix", []string{
    "s3:ObjectCreated:*",
    "s3:ObjectAccessed:*",
    "s3:ObjectRemoved:*",
}, doneCh) {
    if notificationInfo.Err != nil {
        fmt.Println(notificationInfo.Err)
    }
    fmt.Println(notificationInfo)
}
```

7. 客户端自定义设置

SetAppInfo(appName, appVersion string)

给User-Agent添加的自定义应用信息。

参数

参数	类型	描述
appName	string	发请求的应用名称
appVersion	string	发请求的应用版本

示例

```
// Set Application name and version to be used in subsequent API requests.  
minioClient.SetAppInfo("myCloudApp", "1.0.0")
```

SetCustomTransport(customHTTPTransport http.RoundTripper)

重写默认的HTTP transport，通常用于调试或者添加自定义的TLS证书。

参数

参数	类型	描述
customHTTPTransport	http.RoundTripper	自定义的transport，例如：为了调试对API请求响应进行追踪。

TraceOn(outputStream io.Writer)

开启HTTP tracing。追踪信息输出到io.Writer，如果outputstream为nil，则trace写入到os.Stdout标准输出。

参数

参数	类型	描述
outputStream	io.Writer	HTTP trace写入到outputStream

TraceOff()

关闭HTTP tracing。

SetS3TransferAccelerate(acceleratedEndpoint string)

给后续所有API请求设置ASW S3传输加速endpoint。注意：此API仅对AWS S3有效，对其它S3兼容的对象存储服务不生效。

参数

参数	类型	描述
acceleratedEndpoint	string	设置新的S3传输加速endpoint。

适用于与Amazon S3兼容的云存储的MinIO .NET SDK

MinIO .NET Client SDK提供了简单的API来访问MinIO以及任何与Amazon S3兼容的对象存储服务。有关API和示例的完整列表，请查看[Dotnet Client API Reference](#)文档。本文假设你已经有VisualStudio开发环境。

最低需求

- .NET 4.5.2, .NetStandard2.0或更高版本
- Visual Studio 2017

使用NuGet安装

为了安装.NET Framework的MinIO .NET包，你可以在Nuget Package Manager控制台运行下面的命令。

```
PM> Install-Package Minio
```

MinIO Client示例

MinIO client需要以下4个参数来连接与Amazon S3兼容的对象存储服务。

参数	描述
endpoint	对象存储服务的URL
accessKey	Access key是唯一标识你的账户的用户ID。
secretKey	Secret key是你账户的密码。
secure	true代表使用HTTPS。

下面示例中使用运行在 <https://play.min.io> 上的MinIO服务，你可以用这个服务来开发和测试。示例中的访问凭据是公开的。

```
using Minio;

// Initialize the client with access credentials.
private static MinioClient minio = new MinioClient("play.min.io",
    "Q3AM3UQ867SPQQA43P2F",
    "zuf+tfteSlswRu7BJ86wekitnifILbZam1KYY3TG"
).WithSSL();

// Create an async task for listing buckets.
var getListBucketsTask = minio.ListBucketsAsync();

// Iterate over the list of buckets.
foreach (Bucket bucket in getListBucketsTask.Result.Buckets)
{
    Console.WriteLine(bucket.Name + " " + bucket.CreationDateDateTime);
}
```

完整的File Uploader示例

本示例程序连接到一个对象存储服务，创建一个存储桶，并且上传一个文件到该存储桶中。为了运行下面的示例，请点击[\[Link\]](#)启动该项目。

```

using System;
using Minio;
using Minio.Exceptions;
using Minio.DataModel;
using System.Threading.Tasks;

namespace FileUploader
{
    class FileUpload
    {
        static void Main(string[] args)
        {
            var endpoint = "play.min.io";
            var accessKey = "Q3AM3UQ867SPQQA43P2F";
            var secretKey = "zuf+tfteSlswRu7BJ86wekitnifILbZam1KYY3TG";
            try
            {
                var minio = new MinioClient(endpoint, accessKey, secretKey).WithSSL();
                FileUpload.Run(minio).Wait();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
        }

        // File uploader task.
        private async static Task Run(MinioClient minio)
        {
            var bucketName = "mymusic";
            var location = "us-east-1";
            var objectName = "golden-oldies.zip";
            var filePath = "C:\\\\Users\\\\username\\\\Downloads\\\\golden_oldies.mp3";
            var contentType = "application/zip";

            try
            {
                // Make a bucket on the server, if not already present.
                bool found = await minio.BucketExistsAsync(bucketName);
                if (!found)
                {
                    await minio.MakeBucketAsync(bucketName, location);
                }
                // Upload a file to bucket.
                await minio.PutObjectAsync(bucketName, objectName, filePath, contentType);
                Console.WriteLine("Successfully uploaded " + objectName );
            }
            catch (MinioException e)
            {
                Console.WriteLine("File Upload Error: {0}", e.Message);
            }
        }
    }
}

```

运行MinIO Client示例

Windows

- clone这个项目，并在Visual Studio 2017中打开Minio.Sln。

```
$ git clone https://github.com/minio/minio-dotnet && cd minio-dotnet
```

- 在Minio.Examples/Program.cs中输入你的认证信息、存储桶名称、对象名称等。在Program.cs中取消注释以下类似的测试用例来运行示例。

```
//Cases.MakeBucket.Run(minioClient, bucketName).Wait();
```

- 从Visual Studio运行Minio.Client.Examples或

Linux (Ubuntu 16.04)

在Linux上设置Mono和.NETCore

注意：minio-dotnet需要mono 5.0.1稳定版本和.NET Core 2.0 SDK。

- 为你的发行版安装.NETCore和Mono。请参阅示例脚本Ubuntu Xenial [mono_install.sh](#)安装.NETCore和Mono。

```
$ ./mono_install.sh
```

运行Minio.Examples

```
$ cd Minio.Examples  
$ dotnet build -c Release  
$ dotnet run
```

操作存储桶

- [MakeBucket.cs](#)
- [ListBuckets.cs](#)
- [BucketExists.cs](#)
- [RemoveBucket.cs](#)
- [ListObjects.cs](#)
- [ListIncompleteUploads.cs](#)

存储桶策略

- [GetPolicy.cs](#)
- [SetPolicy.cs](#)

存储桶通知

- [GetBucketNotification.cs](#)
- [SetBucketNotification.cs](#)
- [RemoveAllBucketNotifications.cs](#)

操作文件对象

- [FGetObject.cs](#)
- [FPutObject.cs](#)

操作对象

- [GetObject.cs](#)
- [GetPartialObject.cs](#)

- [PutObject.cs](#)
- [StatObject.cs](#)
- [RemoveObject.cs](#)
- [RemoveObjects.cs](#)
- [Object.cs](#)
- [RemoveIncompleteUpload.cs](#)

Presigned操作

- [PresignedGetObject.cs](#)
- [PresignedPutObject.cs](#)
- [PresignedPostPolicy.cs](#)

客户端自定义设置

- [SetAppInfo](#)
- [SetTraceOn](#)
- [SetTraceOff](#)

了解更多

- [完整文档](#)
- [MinIO .NET SDK API文档](#)

.NET Client API参考文档

初始化MinIO Client object。

MinIO

```
var minioClient = new MinioClient("play.min.io",
    "Q3AM3UQ867SPQQA43P2F",
    "zuf+tfteS1swRu7BJ86wekitnifILbZam1KYY3TG"
).WithSSL();
```

AWS S3

```
var s3Client = new MinioClient("s3.amazonaws.com",
    "YOUR-ACCESSKEYID",
    "YOUR-SECRETACCESSKEY"
).WithSSL();
```

操作存储桶	操作对象	Presigned操作	存储桶策略
makeBucket	getObject	presignedGetObject	getBucketPolicy
listBuckets	putObject	presignedPutObject	setBucketPolicy
bucketExists	Object	presignedPostPolicy	setBucketNotification
removeBucket	statObject		getBucketNotification
listObjects	removeObject		removeAllBucketNotification
listIncompleteUploads	removeObjects		
	removeIncompleteUpload		

1. 构造函数

```
public MinioClient(string endpoint, string accessKey = "", string secretKey = "")
```

使用给定的endpoint创建一个MinioClient对象。AccessKey、secretKey和region是可选参数，如果为空的话代表匿名访问。该client对象默认使用HTTP进行访问，如果想使用HTTPS，针对client对象链式调用WithSSL()可启用安全的传输协议。

参数

参数	类型	描述
endpoint	string	endPoint是一个URL，域名，IPv4或者IPv6地址。以下是合法的endpoints: s3.amazonaws.com play.min.io localhost play.min.io
accessKey	string	accessKey类似于用户ID，用于唯一标识你的账户。可选，为空代表匿名访问。
secretKey	string	secretKey是你账户的密码。可选，为空代表匿名访问。
region	string	对象存储的region。可选。

安全访问

```
client对象链式调用.WithSSL(), 可以启用https。
```

示例

MinIO

```
// 1. public MinioClient(String endpoint)
MinioClient minioClient = new MinioClient("play.min.io");

// 2. public MinioClient(String endpoint, String accessKey, String secretKey)
MinioClient minioClient = new MinioClient("play.min.io",
                                         accessKey:"Q3AM3UQ867SPQQA43P2F",
                                         secretKey:"zuf+tfteSlswRu7BJ86wekitnifILbZam1KYY3TG"
                                         ).WithSSL();
```

AWS S3

```
// 1. public MinioClient(String endpoint)
MinioClient s3Client = new MinioClient("s3.amazonaws.com").WithSSL();

// 2. public MinioClient(String endpoint, String accessKey, String secretKey)
MinioClient s3Client = new MinioClient("s3.amazonaws.com:80",
                                         accessKey:"YOUR-ACCESSKEYID",
                                         secretKey:"YOUR-SECRETACCESSKEY").WithSSL();
```

2. 操作存储桶

MakeBucketAsync(string bucketName, string location="us-east-1")

```
Task MakeBucketAsync(string bucketName, string location = "us-east-1", CancellationToken cancellationToken =
default(CancellationToken))
```

创建一个存储桶。

参数

参数	类型	描述
bucketName	string	存储桶名称。
region	string	可选参数。默认是us-east-1。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
Task	列出的异常：
	InvalidBucketNameException :无效的存储桶名称。
	ConnectionException :连接异常。
	AccessDeniedException :拒绝访问。
	RedirectionException :服务器重定向。
	InternalClientException :内部错误。

示例

```
try
{
    // Create bucket if it doesn't exist.
    bool found = await minioClient.BucketExistsAsync("mybucket");
    if (found)
    {
        Console.Out.WriteLine("mybucket already exists");
    }
    else
    {
        // Create bucket 'my-bucketname'.
        await minioClient.MakeBucketAsync("mybucket");
        Console.Out.WriteLine("mybucket is created successfully");
    }
}
catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

ListBucketsAsync()

```
Task<ListAllMyBucketsResult> ListBucketsAsync(CancellationToken cancellationToken = default(CancellationToken))
```

列出所有的存储桶。

参数	类型	描述
cancellationToken	<i>System.Threading.CancellationToken</i>	可选参数。默认是default(CancellationToken)
返回值类型		异常
Task<ListAllMyBucketsResult> : 包含存储桶类型列表的 Task。		列出的异常:
		InvalidOperationException : 无效的存储桶名称。
		ConnectionException : 连接异常。
		AccessDeniedException : 拒绝访问。
		InvalidOperationException : XML数据反序列化异常。
		ErrorResponseException : 执行异常。
		InternalClientException : 内部错误。

示例

```
try
{
    // List buckets that have read access.
    var list = await minioClient.ListBucketsAsync();
    foreach (Bucket bucket in list.Buckets)
    {
        Console.Out.WriteLine(bucket.Name + " " + bucket.CreationDateDateTime);
    }
}
```

```

    }
    catch (MinioException e)
    {
        Console.Out.WriteLine("Error occurred: " + e);
    }
}

```

BucketExistsAsync(string bucketName)

```
Task<bool> BucketExistsAsync(string bucketName, CancellationToken cancellationToken = default(CancellationToken))
```

检查存储桶是否存在。

参数

参数	类型	描述
bucketName	string	存储桶名称。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
Task<bool> : 如果存储桶存在的话则是true。	列出的异常：
	InvalidBucketNameException : 无效的存储桶名称。
	ConnectionException : 连接异常。
	AccessDeniedException : 拒绝访问。
	ErrorResponseException : 执行异常。
	InternalClientException : 内部错误。

示例

```

try
{
    // Check whether 'my-bucketname' exists or not.
    bool found = await minioClient.BucketExistsAsync(bucketName);
    Console.Out.WriteLine("bucket-name " + ((found == true) ? "exists" : "does not exist"));
}
catch (MinioException e)
{
    Console.WriteLine("[Bucket] Exception: {0}", e);
}

```

RemoveBucketAsync(string bucketName)

```
Task RemoveBucketAsync(string bucketName, CancellationToken cancellationToken = default(CancellationToken))
```

删除一个存储桶

注意： - removeBucket不会删除存储桶中的对象，你需要调用removeObject API清空存储桶内的对象。

参数

参数	类型	描述
bucketName	string	存储桶名称。

cancellationToken	<i>System.Threading.CancellationToken</i>	可选参数。默认是default(CancellationToken)
返回值类型	异常	
Task	列出的异常：	
	<code>InvalidBucketNameException</code> ：无效的存储桶名称。	
	<code>ConnectionException</code> ：连接异常。	
	<code>AccessDeniedException</code> ：拒绝访问。	
	<code>ErrorResponseException</code> ：执行异常。	
	<code>InternalClientException</code> ：内部错误。	
	<code>BucketNotFoundException</code> ：存储桶不存在。	

示例

```
try
{
    // Check if my-bucket exists before removing it.
    bool found = await minioClient.BucketExistsAsync("mybucket");
    if (found)
    {
        // Remove bucket my-bucketname. This operation will succeed only if the bucket is empty.
        await minioClient.RemoveBucketAsync("mybucket");
        Console.Out.WriteLine("mybucket is removed successfully");
    }
    else
    {
        Console.Out.WriteLine("mybucket does not exist");
    }
}
catch(MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

ListObjectsAsync(string bucketName, string prefix = null, bool recursive = true)

```
IObservable<Item> ListObjectsAsync(string bucketName, string prefix = null, bool recursive = true, CancellationToken cancellationToken = default(CancellationToken))
```

列出存储桶里的对象。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>prefix</code>	<code>string</code>	对象的前缀。
<code>recursive</code>	<code>bool</code>	<code>true</code> 代表递归查找, <code>false</code> 代表类似文件夹查找, 以"/"分隔, 不查子文件夹。
<code>cancellationToken</code>	<i>System.Threading.CancellationToken</i>	可选参数。默认是default(CancellationToken)

返回值类型	异常

`IObservable<Item>` :an Observable of Items.

`None`

示例

```

try
{
    // Check whether 'mybucket' exists or not.
    bool found = minioClient.BucketExistsAsync("mybucket");
    if (found)
    {
        // List objects from 'my-bucketname'
        Iobservable<Item> observable = minioClient.ListObjectsAsync("mybucket", "prefix", true);
        IDisposable subscription = observable.Subscribe(
            item => Console.WriteLine("OnNext: {0}", item.Key),
            ex => Console.WriteLine("OnError: {0}", ex.Message),
            () => Console.WriteLine("OnComplete: {0}"));
    }
    else
    {
        Console.Out.WriteLine("mybucket does not exist");
    }
}
catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}

```

ListIncompleteUploads(string bucketName, string prefix, bool recursive)

```
Iobservable<Upload> ListIncompleteUploads(string bucketName, string prefix, bool recursive, CancellationToken cancellationToken = default(CancellationToken))
```

列出存储桶中未完整上传的对象。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>prefix</code>	<code>string</code>	对象的前缀。
<code>recursive</code>	<code>bool</code>	<code>true</code> 代表递归查找, <code>false</code> 代表类似文件夹查找, 以"/"分隔, 不查子文件夹。
<code>cancellationToken</code>	<code>System.Threading.CancellationToken</code>	可选参数。默认是 <code>default(CancellationToken)</code>

返回值类型	异常
<code>Iobservable<Upload></code>	<code>None</code>

示例

```

try
{
    // Check whether 'mybucket' exist or not.
    bool found = minioClient.BucketExistsAsync("mybucket");
    if (found)
    {
        // List all incomplete multipart upload of objects in 'mybucket'
    }
}

```

```

IObservable<Upload> observable = minioClient.ListIncompleteUploads("mybucket", "prefix", true);
IDisposable subscription = observable.Subscribe(
    item => Console.WriteLine("OnNext: {0}", item.Key),
    ex => Console.WriteLine("OnError: {0}", ex.Message),
    () => Console.WriteLine("OnComplete: {0}"));
}
else
{
    Console.Out.WriteLine("mybucket does not exist");
}
}
catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}

```

GetPolicyAsync(string bucketName, string objectPrefix)

```
Task<PolicyType> GetPolicyAsync(string bucketName, string objectPrefix, CancellationToken cancellationToken =
default(CancellationToken))
```

获取存储桶或者对象前缀的访问权限。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectPrefix	string	该存储桶下的对象前缀
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
Task<PolicyType> : 指定存储桶和对象前缀的存储桶策略。	列出的异常:
	InvalidBucketNameException : 无效的存储桶名称。
	InvalidObjectPrefixException : 无效的对象前缀。
	ConnectionException : 连接异常。
	AccessDeniedException : 拒绝访问。
	InternalClientException : 内部错误。
	BucketNotFoundException : 存储桶不存在。

示例

```

try
{
    PolicyType policy = await minioClient.GetPolicyAsync("myBucket", objectPrefix:"downloads");
    Console.Out.WriteLine("Current policy: " + policy.GetType().ToString());
}
catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}

```

SetPolicyAsync(string bucketName, string policyJson)

```
Task SetPolicyAsync(string bucketName, string policyJson, CancellationToken cancellationToken = default(CancellationToken))
```

针对存储桶和对象前缀设置访问策略。

参数

参数	类型	描述
bucketName	string	存储桶名称。
PolicyJson	string	要设置的策略。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
Task	列出的异常：
	InvalidBucketNameException : 无效的存储桶名称。
	ConnectionException : 连接异常。
	InternalClientException : 内部错误。
	InvalidBucketNameException : 无效的存储桶名称。

示例

```
try
{
    string policyJson = $@"{{\"Version\":\"2012-10-17\", \"Statement\": [{\"Action\":\"s3:GetBucketLocation\",
    \"Effect\":\"Allow\", \"Principal\":\"*\", \"Resource\":\"arn:aws:s3::{bucketName}\", \"Sid\":\"*\"
    }, {\"Action\":\"s3>ListBucket\", \"Condition\": {\"StringEquals\": {\"s3:prefix\": \"foo\", \"prefix/\"
    \"}}}, {\"Effect\":\"Allow\", \"Principal\":\"*\", \"Resource\":\"arn:aws:s3::{bucketName}\", \"Sid\":\"*\"
    }, {\"Action\":\"s3:GetObject\", \"Effect\":\"Allow\", \"Principal\":\"*\", \"Resource\"
    : \"arn:aws:s3::{bucketName}/foo*\", \"arn:aws:s3::{bucketName}/prefix/*\", \"Sid\":\"*\"
    }]}";
    await minioClient.SetPolicyAsync("myBucket", policyJson);
}
catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

SetBucketNotificationAsync(string bucketName,BucketNotification notification)

```
Task SetBucketNotificationAsync(string bucketName, BucketNotification notification, CancellationToken cancellationToken = default(CancellationToken))
```

给存储桶设置通知。

参数

参数	类型	描述
bucketName	string	存储桶名称。
notification	BucketNotification	要设置的通知。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
-------	----

Task	列出的异常:
	<code>ConnectionException</code> : 连接异常。
	<code>InternalClientException</code> : 内部错误。
	<code>InvalidBucketNameException</code> : 无效的存储桶名称。
	<code>InvalidOperationException</code> : 通知对象序列化异常。

示例

```

try
{
    BucketNotification notification = new BucketNotification();
    Arn topicArn = new Arn("aws", "sns", "us-west-1", "412334153608", "topicminio");

    TopicConfig topicConfiguration = new TopicConfig(topicArn);
    List<EventType> events = new List<EventType>(){ EventType.ObjectCreatedPut , EventType.ObjectCreated };
    topicConfiguration.AddEvents(events);
    topicConfiguration.AddFilterPrefix("images");
    topicConfiguration.AddFilterSuffix("jpg");
    notification.AddTopic(topicConfiguration);

    QueueConfig queueConfiguration = new QueueConfig("arn:aws:sqs:us-west-1:482314153608:testminioqueue1");
    queueConfiguration.AddEvents(new List<EventType>() { EventType.ObjectCreatedCompleteMultipartUpload });
    notification.AddQueue(queueConfiguration);

    await minio.SetBucketNotificationsAsync(bucketName,
                                            notification);
    Console.Out.WriteLine("Notifications set for the bucket " + bucketName + " successfully");
}
catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}

```

GetBucketNotificationAsync(string bucketName)

```
Task<BucketNotification> GetBucketNotificationAsync(string bucketName, CancellationToken cancellationToken = default(CancellationToken))
```

获取存储桶的通知配置。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>cancellationToken</code>	<code>System.Threading.CancellationToken</code>	可选参数。默认是 <code>default(CancellationToken)</code>

返回值类型	异常
<code>Task<BucketNotification></code> : 存储桶的当前通知配置。	列出的异常:
	<code>InvalidBucketNameException</code> : 无效的存储桶名称。
	<code>ConnectionException</code> : 连接异常。
	<code>AccessDeniedException</code> : 拒绝访问。
	<code>InternalClientException</code> : 内部错误。

	BucketNotFoundException : 存储桶不存在。
	InvalidOperationException : xml数据反序列化异常。

示例

```
try
{
    BucketNotification notifications = await minioClient.GetBucketNotificationAsync(bucketName);
    Console.Out.WriteLine("Notifications is " + notifications.ToXML());
}
catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

RemoveAllBucketNotificationsAsync(string bucketName)

```
Task RemoveAllBucketNotificationsAsync(string bucketName, CancellationToken cancellationToken = default(CancellationToken))
```

删除存储桶上所有配置的通知。

参数

参数	类型	描述
bucketName	string	存储桶名称。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
`Task :	列出的异常：
	InvalidBucketNameException : 无效的存储桶名称。
	ConnectionException : 连接异常。
	AccessDeniedException : 拒绝访问。
	InternalClientException : 内部错误。
	BucketNotFoundException : 存储桶不存在。
	InvalidOperationException : xml数据序列化异常。

示例

```
try
{
    await minioClient.RemoveAllBucketNotificationsAsync(bucketName);
    Console.Out.WriteLine("Notifications successfully removed from the bucket " + bucketName);
}
catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

3. 操作对象

GetObjectAsync(string bucketName, string objectName, Action callback)

```
Task GetObjectAsync(string bucketName, string objectName, Action<Stream> callback, CancellationToken cancellationToken = default(CancellationToken))
```

返回对象数据的流。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	存储桶里的对象名称。
callback	Action	处理流的回调函数。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
Task : Task回调，返回含有对象数据的InputStream。	列出的异常：
	InvalidBucketNameException : 无效的存储桶名称。
	ConnectionException : 连接异常。
	InternalClientException : 内部错误。

示例

```
try
{
    // Check whether the object exists using statObject().
    // If the object is not found, statObject() throws an exception,
    // else it means that the object exists.
    // Execution is successful.
    await minioClient.StatObjectAsync("mybucket", "myobject");

    // Get input stream to have content of 'my-objectname' from 'my-bucketname'
    await minioClient.GetObjectAsync("mybucket", "myobject",
        (stream) =>
    {
        stream.To(Console.OpenStandardOutput());
    });
}
catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

GetObjectAsync(string bucketName, string objectName, long offset, long length, Action callback)

```
Task GetObjectAsync(string bucketName, string objectName, long offset, long length, Action<Stream> callback,
CancellationToken cancellationToken = default(CancellationToken))
```

下载对象指定区域的字节数组做为流。offset和length都必须传。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	存储桶里的对象名称。
offset	long	offset 是起始字节的位置。
length	long	length 是要读取的长度。
callback	Action	处理流的回调函数。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是 default(CancellationToken)

返回值类型	异常
Task : Task回调，返回含有对象数据的InputStream。	列出的异常：
	InvalidOperationException : 无效的存储桶名称。
	ConnectionException : 连接异常。
	InternalClientException : 内部错误。

示例

```

try
{
    // Check whether the object exists using statObject().
    // If the object is not found, statObject() throws an exception,
    // else it means that the object exists.
    // Execution is successful.
    await minioClient.StatObjectAsync("mybucket", "myobject");

    // Get input stream to have content of 'my-objectname' from 'my-bucketname'
    await minioClient.GetObjectAsync("mybucket", "myobject", 1024L, 10L,
        (stream) =>
    {
        stream.To(Console.OpenStandardOutput());
    });
}

catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}

```

GetObjectAsync(String bucketName, String objectName, String fileName)

```

Task GetObjectAsync(string bucketName, string objectName, string fileName, CancellationToken cancellationToken = default(CancellationToken))

```

下载并将文件保存到本地文件系统。

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。
fileName	String	本地文件路径。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是 default(CancellationToken)

返回值类型	异常
Task	列出的异常：
	<code>InvalidBucketNameException</code> : 无效的存储桶名称。
	<code>ConnectionException</code> : 连接异常。
	<code>InternalClientException</code> : 内部错误。

示例

```

try
{
    // Check whether the object exists using statObjectAsync().
    // If the object is not found, statObjectAsync() throws an exception,
    // else it means that the object exists.
    // Execution is successful.
    await minioClient.StatObjectAsync("mybucket", "myobject");

    // Gets the object's data and stores it in photo.jpg
    await minioClient.GetObjectAsync("mybucket", "myobject", "photo.jpg");

}
catch (MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}

```

PutObjectAsync(string bucketName, string objectName, Stream data, long size, string contentType)

```
Task PutObjectAsync(string bucketName, string objectName, Stream data, long size, string contentType, Dictionary<string, string> metaData=null, CancellationToken cancellationToken = default(CancellationToken))
```

通过Stream上传对象。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>objectName</code>	<code>string</code>	存储桶里的对象名称。
<code>data</code>	<code>Stream</code>	要上传的Stream对象。
<code>size</code>	<code>long</code>	流的大小。
<code>contentType</code>	<code>string</code>	文件的Content type， 默认是"application/octet-stream"。
<code>metaData</code>	<code>Dictionary</code>	元数据头信息的Dictionary对象， 默认是null。

| `cancellationToken` | `System.Threading.CancellationToken` | 可选参数。默认是`default(CancellationToken)` |

返回值类型	异常
Task	列出的异常：
	<code>InvalidBucketNameException</code> : 无效的存储桶名称。
	<code>ConnectionException</code> : 连接异常。
	<code>InternalClientException</code> : 内部错误。
	<code>EntityTooLargeException</code> : 要上传的大小超过最大允许值。

	UnexpectedShortReadException : 读取的数据大小比指定的size要小。
	ArgumentNullException : Stream为null。

示例

单个对象的最大大小限制在5TB。putObject在对象大于5MiB时，自动使用multiple parts方式上传。这样，当上传失败时，客户端只需要上传未成功的部分即可（类似断点上传）。上传的对象使用MD5SUM签名进行完整性验证。

```
try
{
    byte[] bs = File.ReadAllBytes(fileName);
    System.IO.MemoryStream filestream = new System.IO.MemoryStream(bs);

    await minio.PutObjectAsync("mybucket",
        "island.jpg",
        filestream,
        filestream.Length,
        "application/octet-stream");
    Console.Out.WriteLine("island.jpg is uploaded successfully");
}
catch(MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

PutObjectAsync(string bucketName, string objectName, string filePath, string contentType=null)

```
Task PutObjectAsync(string bucketName, string objectName, string filePath, string contentType=null,Dictionary<string,string> metaData=null, CancellationToken cancellationToken = default(CancellationToken))
```

通过文件上传到对象中。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	存储桶里的对象名称。
fileName	string	要上传的本地文件名。
contentType	string	文件的Content type， 默认是"application/octet-stream"。
metadata	Dictionary	元数据头信息的Dictionary对象， 默认是null。

| cancellationToken | System.Threading.CancellationToken | 可选参数。默认是default(CancellationToken) |

返回值类型	异常
Task	列出的异常：
	InvalidBucketNameException : 无效的存储桶名称。
	ConnectionException : 连接异常。
	InternalClientException : 内部错误。
	EntityTooLargeException : 要上传的大小超过最大允许值。

示例

单个对象的最大大小限制在5TB。putObject在对象大于5MiB时，自动使用multiple parts方式上传。这样，当上传失败时，客户端只需要上传未成功的部分即可（类似断点上传）。上传的对象使用MD5SUM签名进行完整性验证。

```
try
{
    await minio.PutObjectAsync("mybucket", "island.jpg", "/mnt/photos/island.jpg", contentType: "application/octet-stream");
    Console.Out.WriteLine("island.jpg is uploaded successfully");
}
catch(MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

StatObjectAsync(string bucketName, string objectName)

```
Task<ObjectStat> StatObjectAsync(string bucketName, string objectName, CancellationToken cancellationToken =
default(CancellationToken))
```

获取对象的元数据。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	存储桶里的对象名称。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
Task<ObjectStat> : Populated object meta data.	列出的异常：
	InvalidBucketNameException : 无效的存储桶名称。
	ConnectionException : 连接异常。
	InternalClientException : 内部错误。

示例

```
try
{
    // Get the metadata of the object.
    ObjectStat objectStat = await minioClient.StatObjectAsync("mybucket", "myobject");
    Console.Out.WriteLine(objectStat);
}
catch(MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

ObjectAsync(string bucketName, string objectName, string destBucketName, string destObjectName = null, Conditions Conditions = null)

```
Task<ObjectResult> ObjectAsync(string bucketName, string objectName, string destBucketName, string destObjectName
= null, Conditions Conditions = null, CancellationToken cancellationToken = default(CancellationToken))
```

从objectName指定的对象中将数据拷贝到destObjectName指定的对象。

参数

参数	类型	描述
bucketName	string	源存储桶名称。
objectName	string	源存储桶中的源对象名称。
destBucketName	string	目标存储桶名称。
destObjectName	string	要创建的目标对象名称,如果为空, 默认为源对象名称。
Conditions	Conditions	拷贝操作的一些条件Map。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
Task	列出的异常:
	InvalidBucketNameException :无效的存储桶名称。
	ConnectionException :连接异常。
	InternalClientException :内部错误。
	ArgumentException :存储桶不存在。

示例

本API执行了一个服务端的拷贝操作。

```
try
{
    Conditions Conditions = new Conditions();
    Conditions.setMatchETagNone("TestETag");

    await minioClient.ObjectAsync("mybucket", "island.jpg", "mydestbucket", "processed.png", Conditions);
    Console.Out.WriteLine("island.jpg is uploaded successfully");
}
catch(MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

RemoveObjectAsync(string bucketName, string objectName)

```
Task RemoveObjectAsync(string bucketName, string objectName, CancellationToken cancellationToken = default(CancellationToken))
```

删除一个对象。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	存储桶里的对象名称。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
Task	列出的异常:

	<code>InvalidBucketNameException</code> : 无效的存储桶名称。
	<code>ConnectionException</code> : 连接异常。
	<code>InternalClientException</code> : 内部错误。

示例

```
try
{
    // Remove objectname from the bucket my-bucketname.
    await minioClient.RemoveObjectAsync("mybucket", "myobject");
    Console.Out.WriteLine("successfully removed mybucket/myobject");
}
catch (MinioException e)
{
    Console.Out.WriteLine("Error: " + e);
}
```

RemoveObjectAsync(string bucketName, I Enumerable objectsList)

```
Task<IObservable<DeleteError>> RemoveObjectAsync(string bucketName, I Enumerable<string> objectsList, CancellationToken cancellationToken = default(CancellationToken))
```

删除多个对象。

参数

参数	类型	描述
<code>bucketName</code>	<code>string</code>	存储桶名称。
<code>objectsList</code>	<code>IEnumerable</code>	含有多个对象名称的 <code>IEnumerable</code> 。
<code>cancellationToken</code>	<code>System.Threading.CancellationToken</code>	可选参数。默认是 <code>default(CancellationToken)</code>

返回值类型	异常
<code>Task</code>	列出的异常：
	<code>InvalidBucketNameException</code> : 无效的存储桶名称。
	<code>ConnectionException</code> : 连接异常。
	<code>InternalClientException</code> : 内部错误。

示例

```
try
{
    List<String> objectNames = new LinkedList<String>();
    objectNames.add("my-objectname1");
    objectNames.add("my-objectname2");
    objectNames.add("my-objectname3");
    // Remove list of objects in objectNames from the bucket bucketName.
    IObservable<DeleteError> observable = await minio.RemoveObjectAsync(bucketName, objectNames);
    IDisposable subscription = observable.Subscribe(
        deleteError => Console.WriteLine("Object: {0}", deleteError.Key),
        ex => Console.WriteLine("OnError: {0}", ex),
        () =>
    {
        Console.WriteLine("Listed all delete errors for remove objects on " + bucketName + "\n");
    });
}
```

```

    }
    catch (MinioException e)
    {
        Console.Out.WriteLine("Error: " + e);
    }
}

```

RemoveIncompleteUploadAsync(string bucketName, string objectName)

```
Task RemoveIncompleteUploadAsync(string bucketName, string objectName, CancellationToken cancellationToken = default(CancellationToken))
```

删除一个未完整上传的对象。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	存储桶里的对象名称。
cancellationToken	System.Threading.CancellationToken	可选参数。默认是default(CancellationToken)

返回值类型	异常
Task	列出的异常：
	InvalidOperationException : 无效的存储桶名称。
	ConnectionException : 连接异常。
	InternalClientException : 内部错误。

示例

```

try
{
    // Removes partially uploaded objects from buckets.
    await minioClient.RemoveIncompleteUploadAsync("mybucket", "myobject");
    Console.Out.WriteLine("successfully removed all incomplete upload session of my-bucketname/my-objectname"
);
}
catch(MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}

```

4. Presigned操作

PresignedGetObjectAsync(string bucketName, string objectName, int expiresInt, Dictionary reqParams = null);

```
Task<string> PresignedGetObjectAsync(string bucketName, string objectName, int expiresInt, Dictionary<string, string> reqParams = null)
```

生成一个给HTTP GET请求用的presigned URL。浏览器/移动端的客户端可以用这个URL进行下载，即使其所在的存储桶是私有的。这个presigned URL可以设置一个失效时间，默认值是7天。

参数

参数	类型	描述
bucketName	String	存储桶名称。
objectName	String	存储桶里的对象名称。
expiresInt	Integer	失效时间（以秒为单位），默认是7天，不得大于七天。
reqParams	Dictionary	额外的响应头信息，支持response-expire、response-content-type、response-cache-control、response-content-disposition。

返回值类型	异常
Task<string> : string 包含可下载该对象的URL。	列出的异常：
	InvalidBucketNameException : 无效的存储桶名称。
	ConnectionException : 连接异常。
	InvalidExpiryRangeException : 无效的失效时间。

示例

```
try
{
    String url = await minioClient.PresignedGetObjectAsync("mybucket", "myobject", 60 * 60 * 24);
    Console.Out.WriteLine(url);
}
catch(MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}
```

PresignedPutObjectAsync(string bucketName, string objectName, int expiresInt)

```
Task<string> PresignedPutObjectAsync(string bucketName, string objectName, int expiresInt)
```

生成一个给HTTP PUT请求用的presigned URL。浏览器/移动端的客户端可以用这个URL进行上传，即使其所在的存储桶是私有的。这个presigned URL可以设置一个失效时间，默认值是7天。

参数

参数	类型	描述
bucketName	string	存储桶名称。
objectName	string	存储桶里的对象名称。
expiresInt	int	失效时间（以秒为单位），默认是7天，不得大于七天。

返回值类型	异常
Task<string> : string 包含可下载该对象的URL。	列出的异常：
	InvalidBucketNameException : 无效的存储桶名称。
	InvalidKeyException : 无效的access key或secret key。
	ConnectionException : 连接异常。
	InvalidExpiryRangeException : 无效的失效时间。

示例

```

try
{
    String url = await minioClient.PresignedPutObjectAsync("mybucket", "myobject", 60 * 60 * 24);
    Console.Out.WriteLine(url);
}
catch(MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}

```

PresignedPostPolicy(PostPolicy policy)

```
Task<Dictionary<string, string>> PresignedPostPolicyAsync(PostPolicy policy)
```

允许给POST请求的presigned URL设置策略，比如接收对象上传的存储桶名称的策略，key名称前缀，过期策略。

参数

参数	类型	描述
PostPolicy	PostPolicy	对象的post策略。

返回值类型	异常
Task<Dictionary<string, string>> : string的键值对，用于构造表单数据。	列出的异常：
	InvalidBucketNameException : 无效的存储桶名称。
	ConnectionException : 连接异常。
	NoSuchAlgorithmException : 在做签名计算时找不到指定的算法。

示例

```

try
{
    PostPolicy policy = new PostPolicy();
    policy.SetContentType("image/png");
    policy.SetUserMetadata("custom", "user");
    DateTime expiration = DateTime.UtcNow;
    policy.SetExpires(expiration.AddDays(10));
    policy.SetKey("my-objectname");
    policy.SetBucket("my-bucketname");

    Dictionary<string, string> formData = minioClient.Api.PresignedPostPolicy(policy);
    string curlCommand = "curl ";
    foreach (KeyValuePair<string, string> pair in formData)
    {
        curlCommand = curlCommand + " -F " + pair.Key + "=" + pair.Value;
    }
    curlCommand = curlCommand + " -F file=@/etc/bashrc https://s3.amazonaws.com/my-bucketname";
    Console.Out.WriteLine(curlCommand);
}
catch(MinioException e)
{
    Console.Out.WriteLine("Error occurred: " + e);
}

```

Client Custom Settings

SetAppInfo(string appName, string appVersion)

给User-Agent添加应用信息。

参数

参数	类型	描述
appName	string	执行API请求的应用名称。
appVersion	string	执行API请求的应用版本。

示例

```
// Set Application name and version to be used in subsequent API requests.  
minioClient.SetAppInfo("myCloudApp", "1.0.0")
```

SetTraceOn()

开启HTTP tracing, trace日志会输出到stdout。

示例

```
// Set HTTP tracing on.  
minioClient.SetTraceOn()
```

SetTraceOff()

Disables HTTP tracing.

示例

```
// Sets HTTP tracing off.  
minioClient.SetTraceOff()
```

用于Haskell的MinIO Client SDK

MinIO Haskell客户端SDK提供了简单的API，可以访问[MinIO]（<https://min.io>）和与Amazon S3兼容的对象存储服务器。

最低需求

-Haskell 堆栈

安装

添加到您的项目

只需像往常一样，将minio-hs添加到项目的.cabal依赖项部分，或者如果使用hpack，则将其添加到package.yaml文件即可。

直接尝试 ghci

从主文件夹或任何非haskell项目目录中，只需运行：

```
stack install minio-hs
```

然后启动解释器会话，并使用以下命令浏览可用的API：

```
$ stack ghci
> :browse Network.Minio
```

示例

该[示例](#)文件夹中包含许多例子，你可以尝试和使用学习和与发展中国家自己的项目的帮助。

快速入门示例-文件上传器

该示例程序连接到MinIO对象存储服务器，在服务器上创建存储桶，然后将文件上传到存储桶。

在此示例中，我们将使用运行在<https://play.min.io>的MinIO服务器。随意使用此服务进行测试和开发。访问凭证存在于库中，并向公众开放。

FileUploader.hs

```
#!/usr/bin/env stack
-- stack --resolver lts-14.11 runghc --package minio-hs --package optparse-applicative --package filepath

--
-- MinIO Haskell SDK, (C) 2017-2019 MinIO, Inc.
--
-- Licensed under the Apache License, Version 2.0 (the "License");
-- you may not use this file except in compliance with the License.
-- You may obtain a copy of the License at
--
--     http://www.apache.org/licenses/LICENSE-2.0
--
-- Unless required by applicable law or agreed to in writing, software
```

```

-- distributed under the License is distributed on an "AS IS" BASIS,
-- WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
-- See the License for the specific language governing permissions and
-- limitations under the License.

-- 

{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ScopedTypeVariables #-}
import           Network.Minio

import           Data.Monoid          (((<)))
import           Data.Text            (pack)
import           Options.Applicative
import           System.FilePath.Posix
import           UnliftIO              (throwIO, try)

import           Prelude

-- | The following example uses minio's play server at
-- https://play.min.io. The endpoint and associated
-- credentials are provided via the library constant,
--
-- > minioPlayCI :: ConnectInfo
--

-- optparse-applicative package based command-line parsing.
fileNameArgs :: Parser FilePath
fileNameArgs = strArgument
  ( metavar "FILENAME"
    <> help "Name of file to upload to AWS S3 or a MinIO server" )

cmdParser = info
  ( helper <*> fileNameArgs )
  ( fullDesc
    <> progDesc "FileUploader"
    <> header
      "FileUploader - a simple file-uploader program using minio-hs" )

main :: IO ()
main = do
  let bucket = "my-bucket"

  -- Parse command line argument
  filepath <- execParser cmdParser
  let object = pack $ takeBaseName filepath

  res <- runMinio minioPlayCI $ do
    -- Make a bucket; catch bucket already exists exception if thrown.
    bErr <- try $ makeBucket bucket Nothing

    -- If the bucket already exists, we would get a specific
    -- `ServiceErr` exception thrown.
    case bErr of
      Left BucketAlreadyOwnedByYou -> return ()
      Left e                      -> throwIO e
      Right _                     -> return ()

    -- Upload filepath to bucket; object name is derived from filepath.
    fPutObject bucket object filepath defaultPutObjectOptions

  case res of

```

```
Left e  -> putStrLn $ "file upload failed due to " ++ show e  
Right () -> putStrLn "file upload succeeded."
```

运行FileUploader

```
./FileUploader.hs "path/to/my/file"
```

有助于

[贡献者指南](#)

发展历程

建立:

```
git clone https://github.com/minio/minio-hs.git  
cd minio-hs/  
stack install
```

测试可以通过以下方式运行:

```
stack test
```

<https://play.min.io> 默认情况下，测试的一部分使用远程MinIO Play服务器。对于库开发，使用此远程服务器可能很慢。要在本地运行的MinIO live服务器上运行测试 <http://localhost:9000>，只需将环境设置MINIO_LOCAL为任何值（然后取消设置以切换回Play）即可。

要运行实时服务器测试，请设置构建标志，如下所示:

```
stack test --flag minio-hs:live-test  
  
# OR against a local MinIO server with:  
  
MINIO_LOCAL=1 stack test --flag minio-hs:live-test
```

对于每个更改，配置的CI系统始终运行两个测试套件。

可以使用以下方法在本地构建文档:

```
stack haddock
```

MinIO Haskell SDK API参考

初始化**MinIO Client**对象。

MinIO-用于公共Play服务器

```
minioPlayCI :: ConnectInfo  
minioPlayCI
```

AWS S3

```
awsCI :: ConnectInfo  
awsCI { connectAccesskey = "your-access-key"  
       , connectSecretkey = "your-secret-key"  
     }
```

操作存储桶	操作对象	Presigned 操作
listBuckets	getObject	presignedGetObjectUrl
makeBucket	putObject	presignedPutObjectUrl
removeBucket	fGetObject	presignedPostPolicy
listObjects	fPutObject	
listObjectsV1	Object	
listIncompleteUploads	removeObject	
bucketExists	selectObjectContent	

1.连接和运行存储服务上的操作

Haskell MinIO SDK提供了高级功能来执行 MinIO服务器或任何类似于AWS S3的API兼容存储上的操作 服务。

ConnectInfo 类型

“ConnectInfo”记录类型包含一个连接的信息。 特定的服务器。 建议构造 `ConnectInfo` 使用由提供的几个智能构造函数之一的值 库，在以下小节中介绍。

该库通过以下方式自动发现存储区 默认。 这对于可能在其中存储桶的AWS尤其有用 不同地区。 执行上传，下载或其他操作时 操作时，图书馆向服务请求一个位置 存储桶并将其缓存以供后续请求。

awsCI :: ConnectInfo

`awsCI` 是一个提供AWS连接信息的值 S3。 可以通过覆盖两个字段来提供凭据 所以：

```
awsConn = awsCI {  
  connectAccessKey = "my-AWS-access-key"  
  , connectSecretKey = "my-AWS-secret-key"  
}
```

awsWithRegionCI :: Region -> Bool -> ConnectInfo

这个构造函数允许指定初始区域和一个布尔值 启用/禁用自动区域发现行为。

表达式“awsWithRegion region autoDiscover”中的参数为：

参数	类型	描述
region	Region (alias for Text)	默认情况下，所有请求连接到的区域。
autoDiscover	Bool	如果为True，将启用区域发现。如果为False，则禁用发现，并且所有请求仅进入给定区域。

minioPlayCI :: ConnectInfo

该构造函数向以下对象提供连接和身份验证信息 在以下位置连接到公共MinIO Play服务器 <https://play.min.io/> .

minioCI :: Text -> Int -> Bool -> ConnectInfo

用于连接到MinIO服务器。

minioCI host port isSecure 表达式中的参数为：

参数	类型	描述
host	Text	MinIO或其他S3-API兼容服务器的主机名
port	Int	要连接的端口号
isSecure	Bool	服务器是否使用HTTPS？

ConnectInfo字段和默认实例

下表显示了ConnectInfo记录类型中的字段：

字段	类型	描述
connectHost	Text	服务器的主机名。默认为'localhost'。
connectPort	Int	服务器侦听的端口号。默认为9000
connectAccessKey	Text	用于身份验证的访问密钥。默认为 minio。
connectSecretkey	Text	用于身份验证的密钥。默认为 minio123。
connectIsSecure	Bool	指定服务器是否使用TLS。默认为False
connectRegion	Region (alias for Text)	指定要使用的区域。默认为'us-east-1'
connectAutoDiscoverRegion	Bool	指定库是否应自动发现存储桶的区域。默认为True

类型为ConnectInfo的def值具有上述所有默认值 价值观。

The Minio Monad

此monad提供执行请求所需的环境 针对MinIO或其他S3 API兼容服务器。它使用 提供给它的 ConnectInfo 值的连接信息。它 执行连接池，存储桶位置缓存，错误处理 和资源清理行动。

runMinio函数执行Minio中提供的动作 monad并返回一个“ IO (MinioErr a) ”值：

```
{-# Language OverloadedStrings #-}

import Network.Minio

main :: IO ()
main = do
    result <- runMinio def $ do
        buckets <- listBuckets
        return $ length buckets
```

```

case result of
  Left e -> putStrLn $ "Failed operation with error: " ++ show e
  Right n -> putStrLn $ show n ++ " bucket(s) found."

```

上面执行一次“listBuckets”操作并返回服务器中的存储桶。如果有任何错误，将被退回作为类型“MinioErr”的值作为“左”值。

2. 操作存储桶

listBuckets :: Minio [BucketInfo]

列出存储桶。

返回值

返回值类型	描述
<code>Minio [BucketInfo]</code>	列出存储桶

`BucketInfo`记录类型

字段	类型	描述
<code>biName</code>	<code>Bucket (alias of Text)</code>	桶名
<code>biCreationDate</code>	<code>UTCTime</code>	桶的创建时间

makeBucket :: Bucket -> Maybe Region -> Minio ()

创建一个新的存储桶。如果未指定区域，则该区域使用由 `ConnectInfo` 指定的。

参数

在“makeBucket bucketName region”表达式中，参数为：

参数	类型	描述
<code>bucketName</code>	<code>Bucket (alias for Text)</code>	桶名
<code>region</code>	<code>Maybe Region</code>	创建存储桶的区域。如果未指定，则默认为 <code>ConnectInfo</code> 中的区域。

示例

```

{-# Language OverloadedStrings #-}

main :: IO ()
main = do
  res <- runMinio minioPlayCI $ do
    makeBucket bucketName (Just "us-east-1")
  case res of
    Left err -> putStrLn $ "Failed to make bucket: " ++ (show res)
    Right _ -> putStrLn $ "makeBucket successful."

```

removeBucket :: Bucket -> Minio ()

删除存储桶。存储桶必须为空，否则将引发错误。

参数

在表达式 `removeBucket bucketName` 中，参数为：

参数	类型	描述
bucketName	Bucket (alias for Text)	存储桶桶名

示例

```
{-# Language OverloadedStrings #-}

main :: IO ()
main = do
    res <- runMinio minioPlayCI $ do
        removeBucket "mybucket"

    case res of
        Left err -> putStrLn $ "Failed to remove bucket: " ++ (show res)
        Right _ -> putStrLn $ "removeBucket successful."
```

listObjects :: Bucket -> Maybe Text -> Bool -> C.ConduitM () ObjectInfo Minio ()

列出给定存储桶中的对象，实现了AWS S3 API的版本2。

参数

在表达式“listObjects bucketName prefix recursive”中，参数是：

参数	类型	描述
bucketName	Bucket (alias for Text)	存储桶名
prefix	Maybe Text	对象前缀
recursive	Bool	true代表递归查找，false代表类似文件夹查找，以"/"分隔，不查子文件夹。

返回值

返回值类型	描述
C.ConduitM () ObjectInfo Minio ()	对应于每个对象的 ObjectInfo 值的管道生产者。

ObjectInfo记录类型

字段	类型	描述
oiObject	Object (alias for Text)	对象名称
oiModTime	UTCTime	对象的上次修改时间
oiETag	ETag (alias for Text)	对象的ETag
oiSize	Int64	对象的大小（以字节为单位）
oiMetadata	HashMap Text Text	键值用户元数据对的映射

示例

```
{-# LANGUAGE OverloadedStrings #-}
import           Network.Minio

import           Conduit
import           Prelude

-- | The following example uses MinIO play server at
-- https://play.min.io. The endpoint and associated
```

```

-- credentials are provided via the library constant,
--
-- > minioPlayCI :: ConnectInfo
--

main :: IO ()
main = do
  let
    bucket = "test"

  -- Performs a recursive listing of all objects under bucket "test"
  -- on play.min.io.
  res <- runMinio minioPlayCI $
    runConduit $ listObjects bucket Nothing True .| mapM_C (\v -> (liftIO $ print v))
  print res

```

listObjectsV1 :: Bucket -> Maybe Text -> Bool -> C.ConduitM () ObjectInfo Minio ()

列出给定存储桶中的对象，实现AWS S3 API的版本1。这个API提供了与旧版S3兼容的对象存储端点。

参数

在表达式“listObjectsV1 bucketName prefix recursive”中，参数是：

参数	类型	描述
bucketName	Bucket (alias for Text)	存储桶桶名
prefix	Maybe Text	对象前缀
recursive	Bool	true代表递归查找，false代表类似文件夹查找，以"/"分隔，不查子文件夹。

返回值

返回值类型	描述
C.ConduitM () ObjectInfo Minio ()	对应于每个对象的 ObjectInfo 值的管道生产者。

ObjectInfo记录类型

字段	类型	描述
oiObject	Object (alias for Text)	对象名称
oiModTime	UTCTime	对象的上次修改时间
oiETag	ETag (alias for Text)	对象的ETag
oiSize	Int64	对象的大小（以字节为单位）

示例

```

{-# LANGUAGE OverloadedStrings #-}
import           Network.Minio

import           Conduit
import           Prelude

-- | The following example uses MinIO play server at
-- https://play.min.io. The endpoint and associated
-- credentials are provided via the library constant,

```

```

-- > minioPlayCI :: ConnectInfo
--

main :: IO ()
main = do
  let
    bucket = "test"

  -- Performs a recursive listing of all objects under bucket "test"
  -- on play.min.io.
  res <- runMinio minioPlayCI $
    runConduit $ listObjectsV1 bucket Nothing True .| mapM_C (\v -> (liftIO $ print v))
  print res

```

listIncompleteUploads :: Bucket -> Maybe Prefix -> Bool -> C.Producer Minio UploadInfo

列出未完全上传的对象。

参数

在表达式“listIncompleteUploads bucketName前缀递归”中 参数为：

参数	类型	描述
bucketName	Bucket (alias for Text)	存储桶桶名
prefix	Maybe Text	对象前缀
recursive	Bool	true代表递归查找，false代表类似文件夹查找，以'/'分隔，不查子文件夹。

返回值

返回值类型	描述
C.ConduitM () UploadInfo Minio ()	对应于每个不完整分段上传的 UploadInfo 值的管道生产者

UploadInfo记录类型

字段	类型	描述
uiKey	Object	上传对象不完整的名称
uiUploadId	String	未完全上传的对象的上传ID
uiSize	Int64	上传对象不完整的大小

示例

```

{-# LANGUAGE OverloadedStrings #-}
import           Network.Minio

import           Conduit
import           Prelude

-- | The following example uses MinIO play server at
-- https://play.min.io. The endpoint and associated
-- credentials are provided via the library constant,
--
-- > minioPlayCI :: ConnectInfo
-- 

```

```

main :: IO ()
main = do
    let
        bucket = "test"

    -- Performs a recursive listing of incomplete uploads under bucket "test"
    -- on a local MinIO server.
    res <- runMinio minioPlayCI $
        runConduit $ listIncompleteUploads bucket Nothing True .| mapM_C (\v -> (liftIO $ print v))
    print res

```

3. 对象操作

getObject :: Bucket -> Object -> GetObjectOptions -> Minio (C.ConduitM () ByteString Minio ())

从S3服务获取对象，还可以选择提供对象范围。

参数

在表达式“`getObject bucketName objectName opts`”中，参数是：

参数	类型	描述
<code>bucketName</code>	<code>Bucket (alias for Text)</code>	存储桶桶名
<code>objectName</code>	<code>Object (alias for Text)</code>	对象名称
<code>opts</code>	<code>GetObjectOptions</code>	GET请求的选项指定其他选项，例如If-Match, Range

GetObjectOptions记录类型

字段	类型	描述
<code>goRange</code>	<code>Maybe ByteRanges</code>	表示对象的字节范围。例如 <code>ByteRangeFromTo 0 9</code> 表示对象的前十个字节
<code>gooIfMatch</code>	<code>Maybe ETag (alias for Text)</code>	(可选) 对象的ETag应该匹配
<code>gooIfNoneMatch</code>	<code>Maybe ETag (alias for Text)</code>	(可选) 对象的ETag不匹配
<code>gooIfUnmodifiedSince</code>	<code>Maybe UTCTime</code>	(可选) 自对象未被修改以来的时间
<code>gooIfModifiedSince</code>	<code>Maybe UTCTime</code>	(可选) 自修改对象以来的时间

返回值

可以增量读取返回值以处理以下内容 物体。	返回值类型	描述
<code>Minio (C.ConduitM () ByteString Minio ())</code>	字节串值的管道源。	

示例

```

{-# LANGUAGE OverloadedStrings #-}
import Network.Minio

import qualified Data.Conduit      as C
import qualified Data.Conduit.Binary as CB

import Prelude

```

```

-- | The following example uses MinIO play server at
-- https://play.min.io. The endpoint and associated
-- credentials are provided via the library constant,
--
-- > minioPlayCI :: ConnectInfo
--

main :: IO ()
main = do
  let
    bucket = "my-bucket"
    object = "my-object"
  res <- runMinio minioPlayCI $ do
    src <- getObject bucket object def
    C.connect src $ CB.sinkFileCautious "/tmp/my-object"

  case res of
    Left e -> putStrLn $ "getObject failed." ++ (show e)
    Right _ -> putStrLn "getObject succeeded."

```

putObject :: Bucket -> Object -> C.ConduitM () ByteString Minio () -> Maybe Int64 -> PutObjectOptions -> Minio ()

从给定输入将对象上载到服务中的存储桶 具有可选长度的字节流。您也可以选择指定 对象的其他元数据。

参数

在表达式`putObject bucketName objectName inputSrc`中，参数 是：

参数	类型	描述
<code>bucketName</code>	<code>Bucket (alias for Text)</code>	存储桶桶名
<code>objectName</code>	<code>Object (alias for Text)</code>	对象名称
<code>inputSrc</code>	<code>C.ConduitM () ByteString Minio ()</code>	管道生产者的ByteString值
<code>size</code>	<code>Int64</code>	提供流大小（可选）
<code>opts</code>	<code>PutObjectOptions</code>	为对象提供其他元数据的可选参数

示例

```

{-# LANGUAGE OverloadedStrings #-}
import           Network.Minio

import qualified Data.Conduit.Combinators as CC

import           Prelude

-- | The following example uses MinIO play server at
-- https://play.min.io. The endpoint and associated
-- credentials are provided via the library constant,
--
-- > minioPlayCI :: ConnectInfo
--

main :: IO ()
main = do
  let
    bucket = "test"
    object = "obj"

```

```

localFile = "/etc/lsb-release"
kb15 = 15 * 1024

-- Eg 1. Upload a stream of repeating "a" using putObject with default options.
res <- runMinio minioPlayCI $
  putObject bucket object (CC.repeat "a") (Just kb15) def
  case res of
    Left e  -> putStrLn $ "putObject failed." ++ show e
    Right () -> putStrLn "putObject succeeded."

```

fGetObject :: Bucket -> Object -> FilePath -> GetObjectOptions -> Minio ()

将对象从服务中的存储桶下载到给定文件

参数

在表达式“fGetObject bucketName objectName inputFile”中，参数是：

参数	类型	描述
bucketName	Bucket (alias for Text)	存储桶桶名
objectName	Object (alias for Text)	对象名称
inputFile	FilePath	要上传文件的路径
opts	GetObjectOptions	GET请求的选项指定其他选项，例如If-Match, Range

GetObjectOptions记录类型

字段	类型	描述
gooRange	Maybe ByteRanges	表示对象的字节范围。例如ByteRangeFromTo 0 9表示对象的前十个字节
gooIfMatch	Maybe ETag (alias for Text)	(可选) 对象的ETag应该匹配
gooIfNoneMatch	Maybe ETag (alias for Text)	(可选) 对象的ETag不匹配
gooIfUnmodifiedSince	Maybe UTCTime	(可选) 自对象未被修改以来的时间
gooIfModifiedSince	Maybe UTCTime	(可选) 自修改对象以来的时间

```

{-# Language OverloadedStrings #-}

import Network.Minio

import Data.Conduit (( $$+ ))
import Data.Conduit.Binary (sinkLbs)
import Prelude

-- | The following example uses MinIO play server at
-- https://play.min.io. The endpoint and associated
-- credentials are provided via the library constant,
--
-- > minioPlayCI :: ConnectInfo
--

main :: IO ()
main = do
  let
    bucket = "my-bucket"
    object = "my-object"

```

```

localFile = "/etc/lsb-release"

res <- runMinio minioPlayCI $ do
    src <- fGetObject bucket object localFile def
    (src $$+- sinkLbs)

    case res of
        Left e -> putStrLn $ "fGetObject failed." ++ (show e)
        Right _ -> putStrLn "fGetObject succeeded."

```

fPutObject :: Bucket -> Object -> FilePath -> Minio ()

从给定文件将对象上传到服务中的存储桶

参数

在表达式“fPutObject bucketName objectName inputFile”中，参数是：

参数	类型	描述
bucketName	Bucket (alias for Text)	存储桶名称
objectName	Object (alias for Text)	对象名称
inputFile	FilePath	上传文件路径

示例

```

{-# Language OverloadedStrings #-}
import Network.Minio
import qualified Data.Conduit.Combinators as CC

main :: IO ()
main = do
    let
        bucket = "mybucket"
        object = "myobject"
        localFile = "/etc/lsb-release"

    res <- runMinio minioPlayCI $ do
        fPutObject bucket object localFile

    case res of
        Left e -> putStrLn $ "Failed to fPutObject " ++ show bucket ++ "/" ++ show object
        Right _ -> putStrLn "fPutObject was successful"

```

Object :: DestinationInfo -> SourceInfo -> Minio ()

将对象的内容从服务复制到另一个

参数

在表达式Object dstInfo srcInfo中，参数是：

参数	类型	描述
dstInfo	DestinationInfo	代表目标对象属性的值
srcInfo	SourceInfo	代表源对象属性的值

SourceInfo记录类型

字段	类型	描述
srcBucket	Bucket	存储桶名称
srcObject	Object	对象名称
srcRange	Maybe (Int64, Int64)	(可选) 表示源对象的字节范围。 (0, 9) 代表源对象的前十个字节
srcIfMatch	Maybe Text	(可选) ETag源对象应匹配
srcIfNoneMatch	Maybe Text	(可选) ETag源对象不匹配
srcIfUnmodifiedSince	Maybe UTCTime	(可选) 自修改源对象以来的时间
srcIfModifiedSince	Maybe UTCTime	(可选) 自修改源对象以来的时间

Destination记录类型

字段	类型	描述
dstBucket	Bucket	服务器端Object中的目标存储桶名称
dstObject	Object	服务器端Object中目标对象的名称

示例

```
{-# Language OverloadedStrings #-}
import Network.Minio

main :: IO ()
main = do
    let
        bucket = "mybucket"
        object = "myobject"
        object = "obj-"

    res <- runMinio minioPlayCI $ do
        Object def { dstBucket = bucket, dstObject = object } def { srcBucket = bucket, srcObject = object
    }

    case res of
        Left e -> putStrLn $ "Failed to Object " ++ show bucket ++ show "/" ++ show object
        Right _ -> putStrLn "Object was successful"
```

removeObject :: Bucket -> Object -> Minio ()

从服务中删除对象

参数

在表达式 `removeObject bucketName objectName` 中，参数是：

参数	类型	描述
bucketName	Bucket (alias for Text)	存储桶名称
objectName	Object (alias for Text)	对象名称

示例

```
{-# Language OverloadedStrings #-}
import Network.Minio

main :: IO ()
```

```

main = do
  let
    bucket = "mybucket"
    object = "myobject"

    res <- runMinio minioPlayCI $ do
      removeObject bucket object

    case res of
      Left e -> putStrLn $ "Failed to remove " ++ show bucket ++ "/" ++ show object
      Right _ -> putStrLn "Removed object successfully"

```

removeIncompleteUpload :: Bucket -> Object -> Minio ()

从服务中删除正在进行的对象分段上传

参数

在表达式 `removeIncompleteUpload bucketName objectName` 中，参数是：

参数	类型	描述
<code>bucketName</code>	<code>Bucket</code> (alias for <code>Text</code>)	存储桶名称
<code>objectName</code>	<code>Object</code> (alias for <code>Text</code>)	对象名称

示例

```

{-# Language OverloadedStrings #-}
import Network.Minio

main :: IO ()
main = do
  let
    bucket = "mybucket"
    object = "myobject"

    res <- runMinio minioPlayCI $
      removeIncompleteUpload bucket object

    case res of
      Left _ -> putStrLn $ "Failed to remove " ++ show bucket ++ "/" ++ show object
      Right _ -> putStrLn "Removed incomplete upload successfully"

```

selectObjectContent :: Bucket -> Object -> SelectRequest -> Minio (ConduitT () EventMessage Minio ())

从服务中删除正在进行的对象分段上传

参数

在“`selectObjectContent bucketName objectName selReq`”表达式中 参数为：

参数	类型	描述
<code>bucketName</code>	<code>Bucket</code> (alias for <code>Text</code>)	存储桶名称
<code>objectName</code>	<code>Object</code> (alias for <code>Text</code>)	对象名称
<code>selReq</code>	<code>SelectRequest</code>	选择请求参数

`SelectRequest` 记录

该记录是使用`selectRequest`创建的。有关更多信息，请参考[Haddock](#)。

返回值

返回值可用于读取响应中的各个 `EventMessage`。有关更多信息，请参考[Haddock](#)。

返回值	描述
<code>Minio (C.conduitT () EventMessage Minio ())</code>	“EventMessage”值的管道源。

示例

```
{-# Language OverloadedStrings #-}
import Network.Minio

import qualified Conduit           as C

main :: IO ()
main = do
    let
        bucket = "mybucket"
        object = "myobject"

    res <- runMinio minioPlayCI $ do
        let sr = selectRequest "Select * from s3object"
            defaultCsvInput defaultCsvOutput
        res <- selectObjectContent bucket object sr
        C.runConduit $ res C.|| getPayloadBytes C.|| C.stdoutC

    case res of
        Left _ -> putStrLn "Failed!"
        Right _ -> putStrLn "Success!"
```

bucketExists :: Bucket -> Minio Bool

检查存储桶是否存在。

参数

在“`bucketExists bucketName`”表达式中，参数为：

参数	类型	描述
<code>Bucket</code> (alias for <code>Text</code>)		存储桶名称

4. Presigned 操作

presignedGetObjectUrl :: Bucket -> Object -> UrlExpiry -> Query -> RequestHeaders -> Minio ByteString

生成带有身份验证签名的URL，以获取（下载）宾语。在此处传递的所有其他查询参数和标头将是已签名，并且在使用生成的URL时是必需的。询问参数可用于更改由服务器。标头可用于设置Etag匹配条件等。

有关可能的请求参数和标头的列表，请参阅 [GET对象REST API AWS S3](#)文档。

参数

在表达式“`presignedGetObjectUrl bucketName objectName expiry queryParams headers`”中 参数为：

参数	类型	描述

<code>bucketName</code>	<i>Bucket</i> (alias for <code>Text</code>)	存储桶名称
<code>objectName</code>	<i>Object</i> (alias for <code>Text</code>)	对象名称
<code>expiry</code>	<i>UrlExpiry</i> (alias for <code>Int</code>)	网址到期时间 (以秒为单位)
<code>queryParams</code>	<i>Query</i> (from package <code>http-types:Network.HTTP.Types</code>)	查询参数添加到URL
<code>headers</code>	<i>RequestHeaders</i> (from package <code>http-types:Network.HTTP.Types</code>)	网址应使用的请求标头

返回值

返回生成的URL-将包括身份验证信息。

返回值类型	描述
<code>ByteString</code>	生成 <code>presigned URL</code>

示例

```
{-# Language OverloadedStrings #-}

import Network.Minio
import qualified Data.ByteString.Char8 as B

main :: IO ()
main = do
    let
        bucket = "mybucket"
        object = "myobject"

    res <- runMinio minioPlayCI $ do
        -- Set a 7 day expiry for the URL
        presignedGetObjectUrl bucket object (7*24*3600) [] []

        -- Print the URL on success.
        putStrLn $ either
            ("Failed to generate URL: "++) . show)
        B.unpack
        res
```

presignedPutObjectUrl :: Bucket -> Object -> UrlExpiry -> RequestHeaders -> Minio ByteString

生成带有身份验证签名的网址以PUT（上传）宾语。任何多余的标头（如果通过）都将被签名，因此它们是使用URL上载数据时必需。这可以用于例如，在对象上设置用户元数据。

有关可能通过的标头的列表，请参考PUT对象 REST API AWS S3文档。

参数

在表达式“`presignedPutObjectUrl bucketName objectName expiry headers`”中 参数为：

参数	类型	描述
<code>bucketName</code>	<i>Bucket</i> (alias for <code>Text</code>)	存储桶名称
<code>objectName</code>	<i>Object</i> (alias for <code>Text</code>)	对象名称
<code>expiry</code>	<i>UrlExpiry</i> (alias for <code>Int</code>)	网址到期时间 (以秒为单位)
<code>headers</code>	<i>RequestHeaders</i> (from package <code>http-types:Network.HTTP.Types</code>)	网址应使用的请求标头

返回值

返回生成的URL-将包括身份验证 信息。

返回值类型	描述
<code>ByteString</code>	生成 presigned URL

示例

```
{-# Language OverloadedStrings #-}

import Network.Minio
import qualified Data.ByteString.Char8 as B

main :: IO ()
main = do
    let
        bucket = "mybucket"
        object = "myobject"

    res <- runMinio minioPlayCI $ do
        -- Set a 7 day expiry for the URL
        presignedPutObjectUrl bucket object (7*24*3600) [] []

        -- Print the URL on success.
        putStrLn $ either
            ((Failed to generate URL: "++) . show)
            B.unpack
            res
```

presignedPostPolicy :: PostPolicy -> Minio (ByteString, HashMap Text ByteString)

生成预签名的URL和POST策略以通过POST上传文件 请求。 用于浏览器上传并生成表单数据 应当在请求中提交。

PostPolicy参数是使用newPostPolicy函数创建的：

newPostPolicy :: UTCTime -> [PostPolicyCondition] -> Either PostPolicyError PostPolicy

在“ newPostPolicy expirationTime条件”表达式中，参数为：

参数	类型	描述
<code>expirationTime</code>	<code>UTCTime</code> (from package <code>time:Data.Time.UTCTime</code>)	保单的到期时间
<code>conditions</code>	<code>[PostPolicyConditions]</code>	要添加到策略中的条件列表

使用各种辅助功能创建策略条件- 有关详细信息，请参阅Haddock。

由于条件是通过 `newPostPolicy` 验证的，因此它返回一个 任一个值。

返回值

`presignedPostPolicy`返回一个2元组-生成的URL和一个映射 包含应随请求一起提交的表单数据。

示例

```
{-# Language OverloadedStrings #-}

import Network.Minio
```

```

import qualified Data.ByteString      as B
import qualified Data.ByteString.Char8 as Char8
import qualified Data.HashMap.Strict as H
import qualified Data.Text.Encoding  as Enc
import qualified Data.Time          as Time

main :: IO ()
main = do
    now <- Time.getCurrentTime
    let
        bucket = "mybucket"
        object = "myobject"

        -- set an expiration time of 10 days
        expireTime = Time.addUTCTime (3600 * 24 * 10) now

        -- create a policy with expiration time and conditions - since the
        -- conditions are validated, newPostPolicy returns an Either value
        policyE = newPostPolicy expireTime
            [ -- set the object name condition
                ppCondKey "photos/my-object"
                -- set the bucket name condition
                , ppCondBucket "my-bucket"
                -- set the size range of object as 1B to 10MiB
                , ppCondContentLengthRange 1 (10*1024*1024)
                -- set content type as jpg image
                , ppCondContentType "image/jpeg"
                -- on success set the server response code to 200
                , ppCondSuccessActionStatus 200
            ]

    case policyE of
        Left err -> putStrLn $ show err
        Right policy -> do
            res <- runMinio minioPlayCI $ do
                (url, formData) <- presignedPostPolicy policy

                -- a curl command is output to demonstrate using the generated
                -- URL and form-data
                let
                    formFn (k, v) = B.concat ["-F ", Enc.encodeUtf8 k, "=",
                                              " ", v, "\n"]
                    formOptions = B.intercalate " " $ map formFn $ H.toList formData

                return $ B.intercalate " " $
                    ["curl", formOptions, "-F file=@/tmp/photo.jpg", url]

        case res of
            Left e -> putStrLn $ "post-policy error: " ++ (show e)
            Right cmd -> do
                putStrLn $ "Put a photo at /tmp/photo.jpg and run command:\n"
                -- print the generated curl command
                Char8.putStrLn cmd

```


使用S3cmd操作MinIO Server

S3cmd 是用于管理AWS S3, Google云存储或任何使用s3协议的云存储服务提供商的数据的CLI客户端。S3cmd 是开源的，在GPLv2许可下分发。

在本文中，我们将学习如何配置和使用S3cmd来管理MinIO Server的数据。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

2. 安装

从<http://s3tools.org/s3cmd>下载并安装`S3cmd`。

3. 配置

我们将在<https://play.min.io:9000>上运行`S3cmd`。

本示例中的访问凭输入<https://play.min.io:9000>。这些凭据是公开的，你可以随心所欲的使用这个服务来进行测试和开发。[在部署时请替换成你自己的MinIO秘钥](#)。

编辑你的s3cmd配置文件 `~/.s3cfg` 中的以下字段

```
# 设置endpoint
host_base = play.min.io:9000
host_bucket = play.min.io:9000
bucket_location = us-east-1
use_https = True

# 设置access key和secret key
access_key = Q3AM3UQ867SPQQA43P2F
secret_key = zuf+tfteSlswRu7BJ86wekitnifILbZam1KYY3TG

# 启用S3 v4版本签名API
signature_v2 = False
```

4. 命令

创建存储桶

```
s3cmd mb s3://mybucket
Bucket 's3://mybucket/' created
```

拷贝一个文件到存储桶

```
s3cmd put newfile s3://testbucket
upload: 'newfile' -> 's3://testbucket/newfile'
```

拷贝一个文件到本地文件系统

```
s3cmd get s3://testbucket/newfile  
download: 's3://testbucket/newfile' -> './newfile'
```

同步本地文件/文件夹到存储桶

```
s3cmd sync newdemo s3://testbucket  
upload: 'newdemo/newdemofile.txt' -> 's3://testbucket/newdemo/newdemofile.txt'
```

将存储桶或者文件对象同步到本地文件系统

```
s3cmd sync s3://testbucket otherlocalbucket  
download: 's3://testbucket/cat.jpg' -> 'otherlocalbucket/cat.jpg'
```

列举存储桶

```
s3cmd ls s3://  
2015-12-09 16:12 s3://testbbucket
```

列举存储桶里的内容

```
s3cmd ls s3://testbucket/  
DIR s3://testbucket/test/  
2015-12-09 16:05 138504 s3://testbucket/newfile
```

从存储桶里删除一个文件

```
s3cmd del s3://testbucket/newfile  
delete: 's3://testbucket/newfile'
```

删除一个存储桶

```
s3cmd rb s3://mybucket  
Bucket 's3://mybucket/' removed
```

注意: 完整的 `S3cmd` 使用指南可以[在这里](#)找到。

使用AWS CLI结合MinIO Server

AWS CLI是管理AWS服务的统一工具。它通常是用于将数据传入和传出AWS S3的工具。它适用于任何S3兼容的云存储服务。

本文我们将学习如何设置和使用AWS CLI来管理MinIO Server上的数据。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

2. 安装

从<https://aws.amazon.com/cli/>下载安装AWS CLI。

3. 配置

要配置AWS CLI,输入 `aws configure` 并指定MinIO秘钥信息。

本示例中的访问凭据属于<https://play.min.io:9000>。这些凭据是公开的，你可以随心所欲的使用这个服务来进行测试和开发。在部署时请替换成你自己的MinIO秘钥，切记切记切记，重要的事情说三遍。

```
aws configure
AWS Access Key ID [None]: Q3AM3UQ867SPQQA43P2F
AWS Secret Access Key [None]: zuf+tfteSlswRu7BJ86wekitnifILbZam1KYY3TG
Default region name [None]: us-east-1
Default output format [None]: ENTER
```

另外为MinIO Server启用AWS Signature Version'4'。

```
aws configure set default.s3.signature_version s3v4
```

4. 命令

列举你的存储桶

```
aws --endpoint-url https://play.min.io:9000 s3 ls
2016-03-27 02:06:30 deebucket
2016-03-28 21:53:49 guestbucket
2016-03-29 13:34:34 mbtest
2016-03-26 22:01:36 mybucket
2016-03-26 15:37:02 testbucket
```

列举存储桶里的内容

```
aws --endpoint-url https://play.min.io:9000 s3 ls s3://mybucket
2016-03-30 00:26:53      69297 argparse-1.2.1.tar.gz
2016-03-30 00:35:37      67250 simplejson-3.3.0.tar.gz
```

创建一个存储桶

```
aws --endpoint-url https://play.min.io:9000 s3 mb s3://mybucket  
make_bucket: s3://mybucket/
```

往存储桶里添加一个对象

```
aws --endpoint-url https://play.min.io:9000 s3 cp simplejson-3.3.0.tar.gz s3://mybucket  
upload: ./simplejson-3.3.0.tar.gz to s3://mybucket/simplejson-3.3.0.tar.gz
```

从存储桶里删除一个对象

```
aws --endpoint-url https://play.min.io:9000 s3 rm s3://mybucket/argparse-1.2.1.tar.gz  
delete: s3://mybucket/argparse-1.2.1.tar.gz
```

删除一个存储桶

```
aws --endpoint-url https://play.min.io:9000 s3 rb s3://mybucket  
remove_bucket: s3://mybucket/
```

restic结合MinIO Server

restic 是一个快速，高性能，并且安全的备份工具。这是一个在 `BSD 2-Clause License` 下的开源项目。

在本文中，我们将学习如何使用 `restic` 将数据备份到MinIO Server中。

1. 前提条件

从[这里](#)下载并安装MinIO。

2. 安装

从[这里](#)下载并安装restic。

3. 配置

如下所示，在环境变量中设置MinIO认证信息。

```
export AWS_ACCESS_KEY_ID=<YOUR-ACCESS-KEY-ID>
export AWS_SECRET_ACCESS_KEY= <YOUR-SECRET-ACCESS-KEY>
```

4. 命令

启动 `restic`，将指定用于备份的存储桶。

```
./restic -r s3:http://localhost:9000/resticbucket init
```

从本机拷贝需要备份的数据到MinIO Server的存储桶中。

```
./restic -r s3:http://localhost:9000/resticbucket backup /home/minio/workdir/Docs/
enter password for repository:
scan [/home/minio/workdir/Docs]
scanned 2 directories, 6 files in 0:00
[0:00] 100.00% 0B/s 8.045 KiB / 8.045 KiB 6 / 8 items 0 errors ETA 0:00
duration: 0:00, 0.06MiB/s
snapshot 85a9731a saved
```

将MySQL备份存储到MinIO Server

在本文中，我们将学习如何将MySQL备份存储到MinIO Server。

1. 前提条件

- 从[这里](#)下载并安装mc。
- 从[这里](#)下载并安装MinIO Server。
- MySQL官方[文档](#)

2. 配置步骤

MinIO服务正在使用别名 `m1` 运行。从MinIO客户端完整指南[MinIO客户端完全指南](#)了解详情。MySQL备份存储在 `mysqlbkp` 目录下。

创建一个存储桶。

```
mc mb m1/mysqlbkp
Bucket created successfully 'm1/mysqlbkp'.
```

持续地将本地备份文件**mirror**到**MinIO Server**。

持续地将 `mysqlbkp` 文件夹中所有数据**mirror**到MinIO。更多 `mc mirror` 信息，请参考[这里](#)。

```
mc mirror --force --remove --watch mysqlbkp/ m1/mysqlbkp
```

将MongoDB备份存储到MinIO Server

在本文中，我们将学习如何将MongoDB备份存储到MinIO Server。

1. 前提条件

- 从[这里](#)下载并安装mc。
- 从[这里](#)下载并安装MinIO Server。
- MongoDB官方[文档](#)。

2. 配置步骤

MinIO服务正在使用别名 `minio1` 运行。从MinIO客户端完整指南[MinIO客户端完全指南](#)了解详情。MongoDB备份存储在 `mongobkp` 目录下。

创建一个存储桶。

```
mc mb minio1/mongobkp
Bucket created successfully 'minio1/mongobkp'.
```

将**Mongodump**存档流式传输到**MinIO**服务器。

示例中包括w/ SSH tunneling和progress bar。

在一个可信/私有的网络中stream db 'blog-data' :

```
mongodump -h mongo-server1 -p 27017 -d blog-data --archive | mc pipe minio1/mongobkp/backups/mongo-blog-data-
`date +%Y-%m-%d`.archive
```

使用 `--archive` 选项安全地stream 整个 mongodb server。加密备份，我们将 `ssh user@minio-server.example.com` 添加到上面的命令中。

```
mongodump -h mongo-server1 -p 27017 --archive | ssh user@minio-server.example.com mc pipe minio1/mongobkp/full-
1-db-`date +%Y-%m-%d`.archive
```

显示进度和速度信息

我们将添加一个管道到工具 `pv`。（用 `brew install pv` 或 `apt-get install -y pv` 安装）

```
mongodump -h mongo-server1 -p 27017 --archive | pv -brat | ssh user@minio-server.example.com mc pipe minio1/mongobkp/full-db-`date +%Y-%m-%d`.archive
```

持续地将本地备份文件**mirror**到**MinIO Server**。

持续地将 `mongobkp` 文件夹中所有数据**mirror**到MinIO。更多 `mc mirror` 信息，请参考[这里](#)。

```
mc mirror --force --remove --watch mongobkp/ minio1/mongobkp
```


为MinIO Server设置Caddy proxy

Caddy是一个类似于Apache, nginx或者lighttpd的web服务器。Caddy的目的是简化Web开发，部署和托管工作流程，以便任何人都可以托管自己的网站而不需要特殊的技术知识。

在本文中，我们将学习如何给MinIO Server设置Caddy代理。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

2. 安装

从[这里](#)下载并安装Caddy Server。

3. 配置

如下创建Caddy配置文件，根据你的本地minio和DNS配置更改IP地址。

```
your.public.com

proxy / localhost:9000 {
    header_upstream X-Forwarded-Proto {scheme}
    header_upstream X-Forwarded-Host {host}
    header_upstream Host {host}
}
```

4. 步骤

第一步：启动 **minio** 服务。

```
./minio --address localhost:9000 server <your_export_dir>
```

第二步：启动 **caddy** 服务。

```
./caddy
Activating privacy features... done.
your.public.com:443
your.public.com:80
```

为MinIO Server设置Nginx代理

Nginx是一个开源的Web服务器和反向代理服务器。

在本文中，我们将学习如何给MinIO Server设置Nginx代理。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

2. 安装

从[这里](#)安装Nginx。

3. 配置

标准的Root配置

在文件 `/etc/nginx/sites-enabled` 中添加下面的内容，同时删除同一个目录中现有的 `default` 文件。

```
server {
    listen 80;
    server_name example.com;
    location / {
        proxy_set_header Host $http_host;
        proxy_pass http://localhost:9000;
    }
}
```

注意:

- 用你自己的主机名替换example.com。
- 用你自己的服务名替换 `http://localhost:9000`。
- 为了能够上传大文件，在 `http` 上下文中添加 `client_max_body_size 1000m;`，只需按你的需求调整该值。默认值是 `1m`，对大多数场景来说太低了。

非Root配置

当需要非root配置时，按如下方式修改location:

```
location ~^/files {
    proxy_buffering off;
    proxy_set_header Host $http_host;
    proxy_pass http://localhost:9000;
}
```

注意:

- 用你自己的服务名替换 `http://localhost:9000`。
- 用所需的路径替换 `files`。这不能是 `~^/minio`，因为 `minio` 是minio中的保留字。
- 所使用的路径（在本例中为 `files`）按照惯例，应设置为minio所使用的存储桶的名称。
- 可以通过添加更多类似于上面定义的location定义来访问其他存储桶。

使用**Rewrite**的非**Root**配置

以下location配置允许访问任何存储桶，但只能通过未签名的URL，因此只能访问公开的存储桶。

```
location ~^/files {  
    proxy_buffering off;  
    proxy_set_header Host $http_host;  
    rewrite ^/files/(.*)$ /$1 break;  
    proxy_pass http://localhost:9000;  
}
```

注意:

- 用你自己的服务名替换 `http://localhost:9000`。
- 用所需的路径替换 `files`。
- 使用的存储桶必须是公开的，通常情况是可公开读和公开写。
- 使用的网址必须是无符号的，因为nginx会更改网址并使签名无效。

4. 步骤

第一步: 启动**MinIO Server**。

```
minio server /mydatadir
```

第二步: 重启**Nginx server**。

```
sudo service nginx restart
```

了解更多

参考[这里](#)了解更多MinIO和Nginx的配置选项。

使用fluentd插件聚合Apache日志

在本文中，我们将学习如何使用 fluent-plugin-s3 插件结合MinIO做为日志聚合器。

1. 前提条件

- 从[这里](#)下载MinIO Server。
- 从[这里](#)下载 mc。

2. 安装

- 安装并运行[Apache server](#)。
- 安装[fluentd](#) 和 [fluent-plugin-s3](#)。

3. 步骤

第一步：创建存储桶。

fluentd将会实时聚合半结构化apache日志到这个存储桶。

```
mc mb myminio/fluentd
Bucket created successfully 'myminio/fluentd'.
```

第二步：修改fluentd配置以使用MinIO作为存储后端。

将 aws_key_id , aws_sec_key , s3_bucket , s3_endpoint 替换为你自己的值。

将 /etc/td-agent/td-agent.conf 替换为:

```
<source>
@type tail
format apache2
path /var/log/apache2/access.log
pos_file /var/log/td-agent/apache2.access.log.pos
tag s3.apache.access
</source>

<match>
@type s3
aws_key_id `aws_key_id`
aws_sec_key `aws_sec_key`
s3_bucket `s3_bucket`
s3_endpoint `s3_endpoint`
path logs/
force_path_style true
buffer_path /var/log/td-agent/s3
time_slice_format %Y%m%d%H%M
time_slice_wait 10m
utc
buffer_chunk_limit 256m
</match>
```

第三步：重启 fluentd server。

```
sudo /etc/init.d/td-agent restart
```

第四步：检查fluentd的日志以确认是否一切正在运行。

```
tail -f /var/log/td-agent/td-agent.log
path logs/
force_path_style true
buffer_path /var/log/td-agent/s3
time_slice_format %Y%m%d%H%M
time_slice_wait 10m
utc
buffer_chunk_limit 256m
</match>
</ROOT>
2016-05-03 18:44:44 +0530 [info]: following tail of /var/log/apache2/access.log
```

第五步：验证你的配置。

Ping Apache server。该示例采用ab(Apache Bench)程序。

```
ab -n 100 -c 10 http://localhost/
```

第六步：验证聚合的日志。

MinIO Server的fluent存储桶应该显示聚合后的日志。

```
mc ls myminio/fluentd/logs/
[2016-05-03 18:47:13 IST]    570B 201605031306_0.gz
[2016-05-03 18:58:14 IST]    501B 201605031317_0.gz
```

注意事项：

fluentd需要有访问 /var/log/apache2/access.log 的权限。

Rclone结合MinIO Server

Rclone 是一个开源的命令行程序，用来同步文件和目录进或者出云存储系统。它旨在成为"云存储的rsync"。

本文介绍了如何使用rclone来同步MinIO Server。

1. 前提条件

首先从[min.io](#)下载并安装MinIO。

2. 安装

然后从[rclone.org](#)下载并安装Rclone。

3. 配置

当配置好后，MinIO会输出下面的信息

```
Endpoint: http://10.0.0.3:9000 http://127.0.0.1:9000 http://172.17.0.1:9000
AccessKey: USWUXHGYZQYFYFFIT3RE
SecretKey: MOJRH0mkL1IPauahWITSVvyDrQbEEIwljvmxdq03
Region: us-east-1

浏览器访问:
http://10.0.0.3:9000 http://127.0.0.1:9000 http://172.17.0.1:9000

命令行访问: http://docs.minio.org.cn/docs/master/minio-client-quickstart-guide
$ mc config host add myminio http://10.0.0.3:9000 USWUXHGYZQYFYFFIT3RE MOJRH0mkL1IPauahWITSVvyDrQbEEIwljvmxdq03

Object API (Amazon S3 compatible):
Go: http://docs.minio.org.cn/docs/master/golang-client-quickstart-guide
Java: http://docs.minio.org.cn/docs/master/java-client-quickstart-guide
Python: http://docs.minio.org.cn/docs/master/python-client-quickstart-guide
JavaScript: http://docs.minio.org.cn/docs/master/javascript-client-quickstart-guide
```

你现在需要将这些信息配置到rclone。

运行 Rclone config，创建一个新的 s3 类型的remote,叫 minio (你也可以改成别的名字)，然后输入类似下面的信息：

(请注意，按照上面的说明，加入region参数，这很重要。)

```
env_auth> 1
access_key_id> USWUXHGYZQYFYFFIT3RE
secret_access_key> MOJRH0mkL1IPauahWITSVvyDrQbEEIwljvmxdq03
region> us-east-1
endpoint> http://10.0.0.3:9000
location_constraint>
server_side_encryption>
```

配置文件看起来就像这样

```
[minio]
env_auth = false
access_key_id = USWUXHGYZQYFYFFIT3RE
```

```
secret_access_key = MOJRH0mkL1IPauahWITSVvyDrQbEEIwljvmxqdq03F
region = us-east-1
endpoint = http://10.0.0.3:9000
location_constraint =
server_side_encryption =
```

4. 命令

MinIO目前并不支持所有的S3特性。特别是它不支持MD5校验（ETag）或者是元数据。这就表示Rclone不能通过MD5SUMs进行校验或者保存最后修改时间。不过你可以用Rclone的 `--size-only` flag。

下面是一些示例命令

列举存储桶

```
rclone lsd minio:
```

创建一个新的存储桶

```
rclone mkdir minio:bucket
```

拷贝文件到存储桶

```
rclone --size-only /path/to/files minio:bucket
```

从存储桶中拷贝文件

```
rclone --size-only minio:bucket /tmp/bucket-
```

列举存储桶中的所有文件

```
rclone ls minio:bucket
```

同步文件到存储桶 - 先试试 `--dry-run`

```
rclone --size-only --dry-run sync /path/to/files minio:bucket
```

然后再来真的

```
rclone --size-only sync /path/to/files minio:bucket
```

更多示例以及文档，尽在[Rclone web site](#)，不要错过哦。

结合MinIO运行Deis Workflow

Deis Workflow是一个开源的PaaS，可以很容易地在自己的服务器上部署和管理应用程序。Workflow建立于Kubernetes和Docker基础上，提供一个轻量级的PaaS，受Heroku启发的工作流。Workflow有多个模块化比较好的组件（请看<https://github.com/deis>），它们之间使用Kubernetes system和一个对象存储服务进行通信。它有良好的可配置性，可以配置为使用多种云存储服务，像Amazon S3, Google Cloud Storage, Microsoft Azure Storage，当然，还有MinIO。我们目前不会建议你在Deis Workflow生产环境上使用MinIO，目前MinIO可以做为快速安装一个Deis Workflow集群，用于演示、开发、测试的方案。事实上，我们默认提供了装有MinIO的Deis Workflow。

要使用它，请按照<https://docs-v2.readthedocs.io/en/latest/installing-workflow/>中的说明进行操作。完成安装后，请按以下三种方法进行部署：

- Buildpack部署
- Dockerfile部署
- Docker Image部署

所有这三种部署方法，以及Workflow内部广泛使用了MinIO：

- Buildpack部署使用了MinIO存储代码和slugs
- Dockerfile部署使用了MinIO存储Dockerfiles和关联的artifacts
- Docker Image部署使用了MinIO作为运行Workflow的内部Docker registry的后备存储
- Workflow的内部数据库存储用户登录信息，SSH密钥等。它将所有数据备份到MinIO

为MinIO Server设置Apache HTTP proxy

Apache HTTP是一个开源Web服务器和一个反向代理服务器。

在本文中，我们将学习如何使用mod_proxy模块来设置Apache HTTP以连接到MinIO Server。我们将为example.com建立一个新的VirtualHost

1. 前提条件

从[这里](#)下载并安装MinIO Server。记住它的IP和端口。

2. 安装

从[这里](#)安装Apache HTTP server。通常，mod_proxy模块默认是启用的。你也可以使用你的操作系统repositories（例如yum, apt-get）。

3. 步骤

第一步：配置反向代理。

在Apache配置目录下创建一个文件，例如 /etc/httpd/conf.d/minio-vhost.conf

```
<VirtualHost *:80>
    ServerName example.com
    ErrorLog /var/log/httpd/example.com-error.log
    CustomLog /var/log/httpd/example.com-access.log combined

    ProxyRequests Off
    ProxyVia Block
    ProxyPreserveHost On

    <Proxy *>
        Require all granted
    </Proxy>

    ProxyPass / http://localhost:9000/
    ProxyPassReverse / http://localhost:9000/
</VirtualHost>
```

注意：

- 用自己的主机名替换example.com。
- 用自己的服务器名称替换 http://localhost:9000 。

第二步：启动MinIO。

```
minio server /mydatadir
```

第三步：重启Apache HTTP server。

```
sudo service httpd restart
```


使用**pre-signed URLs**通过浏览器上传

使用**presigned URLs**,你可以让浏览器直接上传一个文件到S3服务，而不需要暴露S3服务的认证信息给这个用户。下面就是使用[minio-js](#)实现的一个示例程序。

服务端代码

```
const MinIO = require('minio')

var client = new MinIO.Client({
  endPoint: 'play.min.io',
  port: 9000,
  useSSL: true,
  accessKey: 'Q3AM3UQ867SPQQA43P2F',
  secretKey: 'zuf+tfeS1swRu7BJ86wekitnifILbZam1KYY3TG'
})
```

初始化MinIO client对象，用于生成**presigned upload URL**。

```
// express是一个小巧的Http server封装，不过这对任何HTTP server都管用。
const server = require('express')()

server.get('/presignedUrl', (req, res) => {
  client.presignedPutObject('uploads', req.query.name, (err, url) => {
    if (err) throw err
    res.end(url)
  })
})

server.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
})

server.listen(8080)
```

这里是 [presignedPutObject](#) 的文档。

客户端代码

用户通过浏览器选择了一个文件进行上传，然后在方法内部从Node.js服务端获得了一个URL。然后通过 Fetch API 往这个URL发请求，直接把文件上传到 play.min.io:9000 。

```
<input type="file" id="selector" multiple>
<button onclick="upload()">Upload</button>

<div id="status">No uploads</div>

<script type="text/javascript">
  // `upload` iterates through all files selected and invokes a helper function called `retrieveNewURL`.
  function upload() {
    // Get selected files from the input element.
    var files = document.querySelector("#selector").files;
    for (var i = 0; i < files.length; i++) {
      var file = files[i];
      // 从服务器获取一个URL
```

```

        retrieveNewURL(file, (file, url) => {
            // 上传文件到服务器
            uploadFile(file, url);
        });
    }
}

// 发请求到Node.js server获取上传URL。
// `retrieveNewURL` accepts the name of the current file and invokes the `/presignedUrl` endpoint to
// generate a pre-signed URL for use in uploading that file:
function retrieveNewURL(file, cb) {
    fetch(`/presignedUrl?name=${file.name}`).then((response) => {
        response.text().then((url) => {
            cb(file, url);
        });
    }).catch((e) => {
        console.error(e);
    });
}

// 使用Fetch API来上传文件到S3。
// `uploadFile` accepts the current filename and the pre-signed URL. It then uses `Fetch API`
// to upload this file to S3 at `play.min.io:9000` using the URL:
function uploadFile(file, url) {
    if (document.querySelector('#status').innerText === 'No uploads') {
        document.querySelector('#status').innerHTML = '';
    }
    fetch(url, {
        method: 'PUT',
        body: file
    }).then(() => {
        // If multiple files are uploaded, append upload status on the next line.
        document.querySelector('#status').innerHTML += `  
Uploaded ${file.name}.`;
    }).catch((e) => {
        console.error(e);
    });
}
</script>

```

现在你就可以让别人访问网页，并直接上传文件到S3服务，而不需要暴露S3服务的认证信息。

如何在FreeNAS中运行MinIO

在本文中，我们将学习如何使用FreeNAS运行MinIO。

1. 前提条件

- FreeNAS已经安装并运行,如果没有,请参考[安装说明](#)
- 你有一个FreeNAS Jail path set,如果没有,请参考[jails configuration](#)

2. 安装步骤

创建一个新的Jail

在FreeNAS UI中找到 Jails -> Add Jail , 点击 Advanced , 然后输入如下信息:

```
Name: MinIO
Template: --- (unset, defaults to FreeBSD)
VImage: Unticked
```

为你的环境配置相关的网络设置。点击 OK , 等待Jail下载并安装。

添加存储

找到 Jails -> View Jails -> Storage , 点击 Add Storage , 然后输入如下信息:

```
Jail: MinIO
Source: </path/to/your/dataset>
Destination: </path/to/your/dataset/inside/jail> (usually the same as 'Source' dataset for ease of use)
Read Only: Unticked
Create Directory: Ticked
```

下载MinIO

下载MinIO到jail:

```
curl -Lo <jail_root>/MinIO/usr/local/bin/minio http://dl.minio.org.cn/server/minio/release/freebsd-amd64/minio
chmod +x <jail_root>/MinIO/usr/local/bin/minio
```

创建MinIO服务

创建一个MinIO服务的文件:

```
touch <jail_root>/MinIO/usr/local/etc/rc.d/minio
chmod +x <jail_root>/MinIO/usr/local/etc/rc.d/minio
nano <jail_root>/MinIO/usr/local/etc/rc.d/minio
```

添加下面的内容:

```
#!/bin/sh
# PROVIDE: minio
```

```

# KEYWORD: shutdown

# Define these minio_* variables in one of these files:
#      /etc/rc.conf
#      /etc/rc.conf.local
#      /etc/rc.conf.d/minio
#
# DO NOT CHANGE THESE DEFAULT VALUES HERE
#

# Add the following lines to /etc/rc.conf to enable minio:
#
#minio_enable="YES"
#minio_config="/etc/minio"

minio_enable="${minio_enable-NO}"
minio_config="${minio_config-/etc/minio}"
minio_disks="${minio_disks}"
minio_address="${minio_address-:443}"

. /etc/rc.subr

load_rc_config ${name}

name=minio
rcvar=minio_enable

pidfile="/var/run/${name}.pid"

command="/usr/sbin/demon"
command_args="-c -f -p ${pidfile} /usr/local/bin/${name} -C \"${minio_config}\" server --address=\"${minio_address}\" ${minio_disks}"

run_rc_command "$1"

```

配置MinIO启动

编辑 /<jail_root>/MinIO/etc/rc.conf :

```
nano /<jail_root>/MinIO/etc/rc.conf
```

添加如下内容:

```

minio_enable="YES"
minio_config="/etc/minio"
minio_disks="/path/to/your/dataset/inside/jail"
minio_address="<listen address / port>" (Defaults to :443)

```

创建MinIO配置目录

```
mkdir -p /<jail_root>/MinIO/etc/minio/certs
```

创建MinIO Private key和Public Key (可选,如果需要HTTPS并且 minio_address 设置成443端口)

```
nano <jail_root>/MinIO/etc/minio/certs/public.crt  
nano <jail_root>/MinIO/etc/minio/certs/private.key
```

启动**MinIO Jail**

在FreeNAS UI中找到找到 `Jails -> View Jails` , 选择 `MinIO` , 然后点击 `Start` 按钮 (从左边开始第三个):

测试**MinIO**

找到 `http(s)://<ip_address>:<port>` 并确认**MinIO**加载。

如何使用Cyberduck结合MinIO

在本文档中，你将学习如何使用Cyberduck对MinIO进行基本操作。Cyberduck是适用于MacOS和Windows的FTP和SFTP，WebDAV，OpenStack Swift和Amazon S3的开源客户端。它是在GPL许可证v2.0下发布的。

1. 前提条件

- Cyberduck安装并运行。因为MinIO与Amazon S3兼容，所以你可以从[这里](#)下载一个通用的 HTTP S3 配置文件。
- MinIO Server已经在本地运行，采用 http ,端口9000, 参考 [MinIO快速入门](#)来安装MinIO。

注意: 你也可以用 HTTPS 方式来运行MinIO, 参考[这里](#)，以及[这里](#)描述的Cyberduck通用 HTTPS S3配置文件。

2. 步骤

在**Cyberduck**添加**MinIO**认证信息

点击open connection, 选择 HTTP

修改已有**AWS S3**信息为你本地的**MinIO**凭证

点击**connect**页签建立连接

当连接建立后，那就是天高任鸟飞，你可以去探索更多的操作，下面列出了一部分操作。

列举存储桶

下载存储桶

存储桶镜像

删除存储桶

3. 了解更多

- [MinIO Client完全指南](#)
- [Cyberduck project主页](#)

如何使用AWS SDK for PHP操作MinIO Server

`aws-sdk-php` 是PHP语言版本的官方AWS SDK。本文我们将学习如何使用 `aws-sdk-php` 来操作MinIO Server。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

2. 安装

从[AWS SDK for PHP官方文档](#)下载将安装 `aws-sdk-php`。

3. 使用GetObject和PutObject

下面示例描述的是如何使用`aws-sdk-php`对MinIO Server进行`putObject`和`getObject`操作。请将 `example.php` 文件中的 `endpoint` , `key` , `secret` , `Bucket` 修改为你的本地配置。注意，我们将 `use_path_style_endpoint` 设置为 `true` 以使用 AWS SDK for PHP来操作MinIO。了解更多，请参考[AWS SDK for PHP](#)。

```
<?php

// 使用Composer autoloader引入SDK
date_default_timezone_set('America/Los_Angeles');
require 'vendor/autoload.php';

$s3 = new Aws\S3\S3Client([
    'version' => 'latest',
    'region'  => 'us-east-1',
    'endpoint' => 'http://localhost:9000',
    'use_path_style_endpoint' => true,
    'credentials' => [
        'key'      => 'YOUR-ACCESSKEYID',
        'secret'   => 'YOUR-SECRETACCESSKEY',
    ],
]);

// 发送PutObject请求并获得result对象
$insert = $s3->putObject([
    'Bucket' => 'testbucket',
    'Key'     => 'testkey',
    'Body'    => 'Hello from MinIO!!'
]);

// 下载文件的内容
$retrieve = $s3->getObject([
    'Bucket' => 'testbucket',
    'Key'     => 'testkey',
    'SaveAs'  => 'testkey_local'
]);

// 通过索引到结果对象来打印结果的body。
echo $retrieve['Body'];
```

修改之后，运行程序

```
php example.php
```

```
Hello from MinIO!!
```

4. 生成pre-signed URL

```
<?php
// 从client中获得一个command对象
$command = $s3->getCommand('GetObject', [
    'Bucket' => 'testbucket',
    'Key'      => 'testkey'
]);

// 获得一个10分钟有效期的pre-signed URL
$presignedRequest = $s3->createPresignedRequest($command, '+10 minutes');

// 获得presigned-url
$presignedUrl = (string) $presignedRequest->getUri();
```

5. 获取plain URL

获取一个plain URL,你需要将你的object/bucket权限设为public。注意, 你不会获得带有后面这些信息的URL, X-Amz-Algorithm=[...]&X-Amz-Credential=[...]&X-Amz-Date=[...]&X-Amz-Expires=[...]&X-Amz-SignedHeaders=[...]&X-Amz-Signature=[...]

```
<?php
$plainUrl = $s3->getObjectUrl('testbucket', 'testkey');
```

6. 设置存储桶策略

```
<?php
$bucket = 'testbucket';
// 该策略设置存储桶为只读
$policyReadOnly = '{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:GetBucketLocation",
                "s3>ListBucket"
            ],
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "*"
                ]
            },
            "Resource": [
                "arn:aws:s3:::%s"
            ],
            "Sid": ""
        },
        {
            "Action": [
                "s3:GetObject"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:s3:::%s/*"
            ]
        }
    ]
}'
```

```
"Principal": {
    "AWS": [
        "*"
    ],
},
"Resource": [
    "arn:aws:s3:::%s/*"
],
"Sid": ""
}
];
// 如果你想将文件放到指定目录，你只需要修改'arn:aws:s3:::%s/*'为'arn:aws:s3:::%s/folder/*'

// 创建一个存储桶
$result = $s3->createBucket([
    'Bucket' => $bucket,
]);
// 配置策略
$s3->putBucketPolicy([
    'Bucket' => $bucket,
    'Policy' => sprintf($policyReadOnly, $bucket, $bucket),
]);

```

如何使用AWS SDK for Ruby操作MinIO Server

`aws-sdk` for Ruby是Ruby语言版本的官方AWS SDK。本文我们将学习如何使用 `aws-sdk` for Ruby来操作MinIO Server。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

2. 安装

从[AWS SDK for Ruby官方文档](#)下载将安装 `aws-sdk` for Ruby。

3. 示例

修改 `example.rb` 文件中的 `endpoint` , `access_key_id` , `secret_access_key` , `Bucket` 以及 `Object` 配置成你的本地配置。

下面示例描述的是如何使用 `aws-sdk` for Ruby从MinIO Server上执行`put_object()`和`get_object()`。

```
require 'aws-sdk'

Aws.config.update(
  endpoint: 'http://localhost:9000',
  access_key_id: 'YOUR-ACCESSKEYID',
  secret_access_key: 'YOUR-SECRETACCESSKEY',
  force_path_style: true,
  region: 'us-east-1'
)

rubys3_client = Aws::S3::Client.new

# put_object操作

rubys3_client.put_object(
  key: 'testobject',
  body: 'Hello from MinIO!!!',
  bucket: 'testbucket',
  content_type: 'text/plain'
)

# get_object操作

rubys3_client.get_object(
  bucket: 'testbucket',
  key: 'testobject',
  response_target: 'download_testobject'
)

print "Downloaded 'testobject' as 'download_testobject'. "
```

4. 运行程序

```
ruby example.rb
Downloaded 'testobject' as 'download_testobject'.
```


如何使用AWS SDK for Python操作MinIO Server

`aws-sdk-python` 是Python语言版本的官方AWS SDK。本文我们将学习如何使用 `aws-sdk-python` 来操作MinIO Server。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

2. 安装

从[AWS SDK for Python官方文档](#)下载将安装 `aws-sdk-python`。

3. 示例

修改 `example.py` 文件中的 `endpoint_url` , `aws_access_key_id` , `aws_secret_access_key` , `Bucket` 以及 `Object` 配置成你的本地配置。

下面的示例讲的是如何使用 `aws-sdk-python` 从MinIO Server上进行上传和下载。

```
#!/usr/bin/env/python
import boto3
from botocore.client import Config

s3 = boto3.resource('s3',
                    endpoint_url='http://localhost:9000',
                    aws_access_key_id='YOUR-ACCESSKEYID',
                    aws_secret_access_key='YOUR-SECRETACCESSKEY',
                    config=Config(signature_version='s3v4'),
                    region_name='us-east-1')

# 上传本地文件'/home/john/piano.mp3'到存储桶'songs'，以'piano.mp3'做为object name。
s3.Bucket('songs').upload_file('/home/john/piano.mp3','piano.mp3')

# 从存储桶'songs'里下载文件'piano.mp3'，并保存成本地文件/tmp/classical.mp3
s3.Bucket('songs').download_file('piano.mp3', '/tmp/classical.mp3')

print "Downloaded 'piano.mp3' as 'classical.mp3'. "
```

4. 运行程序

```
python example.py
Downloaded 'piano.mp3' as 'classical.mp3'.
```

5. 了解更多

- [MinIO Python Library for Amazon S3](#)

如何使用**Mountain Duck**结合**MinIO**

在本文中，你将学习如何使用**Mountain Duck**（中文名是山鸭，不是山鸡）在**MinIO**上进行基本操作。**Mountain Duck**可让你将服务器和云存储装载为Mac上的Finder.app和Windows上的文件资源管理器中的本地磁盘。它是在GPL许可证v2.0下发布的。

1. 前提条件

- **Mountain Duck**已经安装并运行。由于**MinIO**与Amazon S3兼容,请从[这里](#)下载一个通用的 HTTP S3配置文件。
- **MinIO Server**已经在本地运行，采用 `http` ,端口9000, 参考[MinIO快速入门](#)来安装**MinIO**。

注意: 你也可以用 `HTTPS` 方式来运行**MinIO**, 参考[这里](#)，以及[这里](#)描述的**Mountain Duck**通用 `HTTPS` S3配置文件。

2. 步骤

在**Mountain Duck**添加**MinIO**认证信息

点击**Mountain Duck**图标，通过导航菜单打开应用程序。点击打开连接，选择 `S3(HTTP)`

修改已有**AWS S3**信息为你本地的**MinIO**凭证

点击**connect**页签建立连接

你将被要求连接通过不安全的连接，因为我们使用**HTTP**而不是**HTTPS**，接受它。建立连接后，你可以进一步探索，下面列出了一些操作。

列举存储桶

复制存储桶到本地文件系统

删除存储桶

3. 了解更多

- [MinIO Client完全指南](#)
- [Mountain Duck project homepage](#)

如何使用AWS SDK for Javascript操作MinIO Server

本文我们将学习如何使用 aws-sdk for Javascript操作MinIO Server。aws-sdk for Javascript是Javascript语言版本的官方AWS SDK。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

2. 安装

从 [AWS Javascript SDK官方文档](#)下载将安装 aws-sdk for Javascript。

3. 示例

修改 example.js 文件中的 endpoint , accessKeyId , secretAccessKey , Bucket 以及 Object 配置成你的本地配置。

下面的示例讲的是如何使用 aws-sdk for Javascript从MinIO Server上putObject和getObject。

```
var AWS = require('aws-sdk');

var s3 = new AWS.S3({
    accessKeyId: 'YOUR-ACCESSKEYID',
    secretAccessKey: 'YOUR-SECRETACCESSKEY',
    endpoint: 'http://127.0.0.1:9000',
    s3ForcePathStyle: true,
    signatureVersion: 'v4'
});

// putObject操作

var params = {Bucket: 'testbucket', Key: 'testobject', Body: 'Hello from MinIO!!'};

s3.putObject(params, function(err, data) {
    if (err)
        console.log(err)
    else
        console.log("Successfully uploaded data to testbucket/testobject");
});

// getObject操作

var params = {Bucket: 'testbucket', Key: 'testobject'};

var file = require('fs').createWriteStream('/tmp/mykey');

s3.getObject(params).
on('httpData', function(chunk) { file.write(chunk); }).
on('httpDone', function() { file.end(); }).
send();
```

4. 运行程序

```
node example.js
```

```
Successfully uploaded data to testbucket/testobject
```

5. 了解更多

- [Javascript Shopping App](#)

如何使用**Træfik**代理多个**MinIO**服务

Træfik是一个用Go语言写的先进（和流行技术结合的比较好）的反向代理。它支持多种配置方式，本文将介绍如何通过Docker设置多个MinIO实例，并用**Træfik**可实现通过不同的子域名进行访问。

1. 前提条件

已经安装Docker并运行，如果没有参考[安装说明](#)。

2. 步骤

获取，配置和启动**Træfik**

首先你应该为**Træfik**创建一个配置文件来启用Let's Encrypt并配置Docker后端。通过HTTP获取请求自动重定向到HTTPS，证书通过集成的Let's Encrypt进行创建。

```
cat << EOF > traefik.toml
defaultEntryPoints = ["http", "https"]

[entryPoints]
[entryPoints.http]
address = ":80"
[entryPoints.http.redirect]
entryPoint = "https"
[entryPoints.https]
address = ":443"
[entryPoints.https.tls]

[acme]
email = "your@email.com"
storageFile = "/etc/traefik/acme.json"
entryPoint = "https"
onDemand = true

[docker]
endpoint = "unix:///var/run/docker.sock"
domain = "example.com"
watch = true
EOF
```

除了上面的配置之外，我们还需要touch一下 `acme.json`，这个文件存了生成的证书，同时也存着私钥，所以你需要设置好权限，别让所有人都能访问这个文件。

```
touch acme.json
chmod 640 acme.json
```

经过上述步骤，我们已经准备好了可以代理请求的**Træfik**容器。

```
docker run -d \
--restart always \
--name traefik \
--publish 80:80 \
--publish 443:443 \
--volume ${pwd}/traefik.toml:/etc/traefik/traefik.toml \
```

```
--volume $(pwd)/acme.json:/etc/traefik/adme.json \
--volume /var/run/docker.sock:/var/run/docker.sock \
traefik
```

获取，配置和启动MinIO

现在咱们可以准备多个MinIO的实例，来演示一个多租户场景的解决方案。你可以启动多个MinIO实例，让Traefik基于不同的凭据信息来进行路由。

我们将从宿主机上启动多个带有挂载卷的MinIO实例。如果你更喜欢data containers，请参考[MinIO Docker 快速入门](#)。

```
for i in $(seq 1 5); do
    mkdir -p $(pwd)/minio${i}/{export,config}

    docker run -d \
        --restart always \
        --name minio-${i} \
        --volume $(pwd)/minio${i}/config:/root/.minio \
        --volume $(pwd)/minio${i}/export:/export \
        minio/minio
done
```

测试启动的实例

你可以用curl来测试启动的实例，这样你就可以确认实例是否启动正确。

```
curl -H Host:minio-1.example.com http://127.0.0.1
```

这个请求会获得下面的输出信息，因为没有认证，不过你可以看到确实是正确启动了。

```
<?xml version="1.0" encoding="UTF-8"?>
<Error><Code>AccessDenied</Code><Message>Access Denied.</Message><Key></Key><BucketName></BucketName><Resource></Resource><RequestId>3L137</RequestId><HostId>3L137</HostId></Error>
```

现在你可以通过 <https://minio-{1,2,3,4,5}.example.com> 来访问所有的MinIO实例。

最后我想多说一句，你应该用你操作系统的init system来启支MinIO的Docker容器。做为示例，你可以看到我是如何使用systemd service来启动新的MinIO实例。就是把这个文件存成 /etc/systemd/system/minio@.service，并且用systemctl start minio@server1 来启动新的实例，然后这个实例就可以通过 server1.example.com 来访问了，诸如此类。

```
[Unit]
Description=MinIO: %i

Requires=docker.service
After=docker.service

[Service]
Restart=always

ExecStop=/bin/sh -c '/usr/bin/docker ps | /usr/bin/grep %p-%i 1> /dev/null && /usr/bin/docker stop %p-%i || true'
ExecStartPre=/bin/sh -c '/usr/bin/docker ps | /usr/bin/grep %p-%i 1> /dev/null && /usr/bin/docker kill %p-%i || true'
ExecStartPre=/bin/sh -c '/usr/bin/docker ps -a | /usr/bin/grep %p-%i 1> /dev/null && /usr/bin/docker rm %p-%i || true'
ExecStartPre=/usr/bin/docker pull minio/minio:latest
```

```
ExecStart=/usr/bin/docker run --rm \
--name %p-%i \
--volume /storage/%p/%i/files:/export \
--volume /storage/%p/%i/config:/root/.minio \
--label traefik.frontend.rule=Host:%i.example.com \
--label traefik.frontend.passHostHeader=true \
minio/minio:latest

[Install]
WantedBy=multi-user.target
```

如何使用AWS SDK for Go操作MinIO Server

`aws-sdk-go` 是GO语言版本的官方AWS SDK。本文将学习如何使用 `aws-sdk-go` 来操作MinIO Server。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

2. 安装

从[AWS SDK for GO官方文档](#)下载将安装 `aws-sdk-go`。

3. 示例

替换 `example.go` 文件中的 `Endpoint` , `Credentials` , `Bucket` 配置成你的本地配置。

下面的示例讲的是如何使用`aws-sdk-go`从MinIO Server上`putObject`和`getObject`。

```
package main

import (
    "fmt"
    "os"
    "strings"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/credentials"
    "github.com/aws/aws-sdk-go/service/s3/s3manager"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)

func main() {
    bucket := aws.String("newbucket")
    key := aws.String("testobject")

    // 配置成使用MinIO Server。
    s3Config := &aws.Config{
        Credentials: credentials.NewStaticCredentials("YOUR-ACCESSKEYID", "YOUR-SECRETACCESSKEY", ""),
        Endpoint:     aws.String("http://localhost:9000"),
        Region:      aws.String("us-east-1"),
        DisableSSL:   aws.Bool(true),
        S3ForcePathStyle: aws.Bool(true),
    }
    newSession := session.New(s3Config)

    s3Client := s3.New(newSession)

    cparams := &s3.CreateBucketInput{
        Bucket: bucket, // 必须
    }

    // 调用CreateBucket创建一个新的存储桶。
    _, err := s3Client.CreateBucket(cparams)
    if err != nil {
        // 错误信息
    }
}
```

```

        fmt.Println(err.Error())
        return
    }

    // 上传一个新的文件"testobject"到存储桶"newbucket", 内容是"Hello World!"。
    _, err = s3Client.PutObject(&s3.PutObjectInput{
        Body:   strings.NewReader("Hello from MinIO!!"),
        Bucket: bucket,
        Key:    key,
    })
    if err != nil {
        fmt.Printf("Failed to upload data to %s/%s, %s\n", *bucket, *key, err.Error())
        return
    }
    fmt.Printf("Successfully created bucket %s and uploaded data with key %s\n", *bucket, *key)

    // 从 "newbucket"里获取文件"testobject", 并保存到本地文件"testobject_local"。
    file, err := os.Create("testobject_local")
    if err != nil {
        fmt.Println("Failed to create file", err)
        return
    }
    defer file.Close()

    downloader := s3manager.NewDownloader(newSession)
    numBytes, err := downloader.Download(file,
        &s3.GetObjectInput{
            Bucket: bucket,
            Key:    key,
        })
    if err != nil {
        fmt.Println("Failed to download file", err)
        return
    }
    fmt.Println("Downloaded file", file.Name(), numBytes, "bytes")
}

```

4. 运行程序

```

go run example.go
Successfully created bucket newbucket and uploaded data with key testobject
Downloaded file testobject_local 18 bytes

```

如何使用AWS SDK for Java操作MinIO Server

`aws-sdk-java` 是Java语言版本的官方AWS SDK。本文我们将学习如何使用 `aws-sdk-java` 操作MinIO Server。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

2. 设置依赖

你可以参考[AWS Java SDK文档](#)下载并安装[aws-java-sdk](#)，或者使用Apache Maven来获得相应的依赖。

```
...
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <aws.sdk.version>1.11.106</aws.sdk.version>
</properties>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-java-sdk-bom</artifactId>
            <version>${aws.sdk.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-java-sdk-s3</artifactId>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
...
```

3. 示例

使用下面的代码替换 `aws-java-sdk-1.11.213/samples/AmazonS3/S3Sample.java`，并且修改 `Endpoint`，`BasicAWSCredentials`，`bucketName`，`uploadFileName` 和 `keyName` 成你的本地配置。

下面的示例描述的是如何使用asw-sdk-java来对MinIO Server进行上传和下载操作。

```
import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWS Credentials;
import com.amazonaws.auth.AWSStaticCredentialsProvider;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.GetObjectRequest;
import com.amazonaws.services.s3.model.PutObjectRequest;
import com.amazonaws.services.s3.model.S3Object;

public class S3Sample {
    private static String bucketName = "testbucket";
    private static String keyName = "hosts";
    private static String uploadFileName = "/etc/hosts";

    public static void main(String[] args) throws IOException {
        AWS Credentials credentials = new BasicAWSCredentials("YOUR-ACCESSKEYID", "OUR-SECRETACCESSKEY");
        ClientConfiguration clientConfiguration = new ClientConfiguration();
        clientConfiguration.setSignerOverride("AWS S3 V4 Signer Type");

        AmazonS3 s3Client = AmazonS3ClientBuilder
            .standard()
            .withEndpointConfiguration(new AwsClientBuilder.EndpointConfiguration("http://localhost:9000"
, Regions.US_EAST_1.name()))
            .withPathStyleAccessEnabled(true)
            .withClientConfiguration(clientConfiguration)
            .withCredentials(new AWSStaticCredentialsProvider(credentials))
            .build();

        try {
            System.out.println("Uploading a new object to S3 from a file\n");
            File file = new File(uploadFileName);
            // 上传文件
            s3Client.putObject(new PutObjectRequest(bucketName, keyName, file));

            // 下载文件
            GetObjectRequest rangeObjectRequest = new GetObjectRequest(bucketName, keyName);
            S3Object objectPortion = s3Client.getObject(rangeObjectRequest);
            System.out.println("Printing bytes retrieved:");
            displayTextInputStream(objectPortion.getObjectContent());
        } catch (AmazonServiceException ase) {
            System.out.println("Caught an AmazonServiceException, which " + "means your request made it "
                + "to Amazon S3, but was rejected with an error response" + " for some reason.");
            System.out.println("Error Message: " + ase.getMessage());
            System.out.println("HTTP Status Code: " + ase.getStatusCode());
            System.out.println("AWS Error Code: " + ase.getErrorCode());
            System.out.println("Error Type: " + ase.getErrorType());
            System.out.println("Request ID: " + ase.getRequestId());
        }
    }
}
```

```

        } catch (AmazonClientException ace) {
            System.out.println("Caught an AmazonClientException, which " + "means the client encountered " +
"an internal error while trying to "
                + "communicate with S3, " + "such as not being able to access the network.");
            System.out.println("Error Message: " + ace.getMessage());
        }
    }

private static void displayTextInputStream(InputStream input) throws IOException {
    // 按行读取并打印。
    BufferedReader reader = new BufferedReader(new InputStreamReader(input));
    while (true) {
        String line = reader.readLine();
        if (line == null)
            break;

        System.out.println("    " + line);
    }
    System.out.println();
}
}

```

4. 运行程序

```

ant
Buildfile: /home/ubuntu/aws-java-sdk-1.11.213/samples/AmazonS3/build.xml

run:
[java] Uploading a new object to S3 from a file
[java]
[java] Printing bytes retrieved:
[java]     127.0.0.1      localhost
[java]     127.0.1.1      minio
[java]
[java]     # 对于支持IPv6的主机，会有以下几行输出。
[java]     ::1      localhost ip6-localhost ip6-loopback
[java]     ff02::1 ip6-allnodes
[java]     ff02::2 ip6-allrouters
[java]

BUILD SUCCESSFUL
Total time: 3 seconds

```

5. 了解更多

- [MinIO Java Library for Amazon S3](#)

如何使用Paperclip操作MinIO Server

Paperclip旨在作为一个简单的ActiveRecord文件附件库。在本文中，你将学习如何将MinIO配置为Paperclip的对象存储后端。

1. 前提条件

MinIO Server已经安装并运行。参考[这里](#)下载并安装MinIO Server。

本文使用<https://play.min.io:9000>。Play(demo Version)做为托管MinIO Server,仅用于进行测试和开发。Play使用
access_key_id Q3AM3UQ867SPQQA43P2F , secret_access_key zuf+tfteS1swRu7BJ86wekitnifILbZam1KYY3TG 。

2. 安装

从[这里](#)安装Paperclip

3. Paperclip存储配置

```
config.paperclip_defaults = {
  storage: :s3,
  s3_protocol: ':https',
  s3_permissions: 'public',
  s3_region: 'us-east-1',
  s3_credentials: {
    bucket: 'mytestbucket',
    access_key_id: 'Q3AM3UQ867SPQQA43P2F',
    secret_access_key: 'zuf+tfteS1swRu7BJ86wekitnifILbZam1KYY3TG',
  },
  s3_host_name: 'play.min.io:9000',
  s3_options: {
    endpoint: "https://play.min.io:9000",
    force_path_style: true
  },
  url: ':s3_path_url',
  path: "/:class/:id.:style.:extension"
}
```

4. 了解更多

[MinIO Paperclip Application](#)

如何使用AWS SDK for .NET操作MinIO Server

`aws-sdk-dotnet` 是.NET Framework的官方AWS开发工具包。在本文中，我们将学习如何使用`aws-sdk-dotnet`来操作MinIO Server。

1. 前提条件

从[这里](#)下载并安装MinIO Server。

如果在本地运行MinIO Server,必须要设置`MINIO_REGION`环境变量。

安装Visual Studio 2015, Visual Studio 2017或者Visual Studio Code。从[这里](#)查找Visual Studio的安装版本。

2. 安装

`aws-sdk-dotnet` 有[Nuget](#)包。此软件包仅包含使用AWS S3所必需的库。Nuget包的安装可以使用"Manage Nuget Packages..." UI, 或者使用Nuget管理控制台, 输入`Install-Package AWSSDK.S3`。安装程序将自动下载与你的项目兼容的.NET平台的库。.NET Frameworks 3.5、4.5 以及 .NET Core 1.1都有这个包。

老版本（version 2）的包在[这里](#), 不过不建议使用, 因为它会下载所有的AWS SDK库, 而不是只下载S3模块。

3. 示例

下面示例的代码应该直接复制, 而不是用自动生成的`Program.cs`文件里的代码。在Visual Studio IDE中创建一个控制台项目, 并用下面的代码替换生成的`Program.cs`。更新`ServiceURL`, `accessKey` 和 `secretKey` 成为你的MinIO Server的配置。

下面的示例采用`aws-sdk-dotnet`以列举的方式打印出MinIO Server里所有的存储桶和第一个存储桶里的所有对象。

```
using Amazon.S3;
using System;
using System.Threading.Tasks;
using Amazon;

class Program
{
    private const string accessKey = "PLACE YOUR ACCESS KEY HERE";
    private const string secretKey = "PLACE YOUR SECRET KEY HERE"; // 不要把你的秘钥硬编码到你的代码中。

    static void Main(string[] args)
    {
        Task.Run(MainAsync).GetAwaiter().GetResult();
    }

    private static async Task MainAsync()
    {
        var config = new AmazonS3Config
        {
            RegionEndpoint = RegionEndpoint.USEast1, // 必须在设置ServiceURL前进行设置, 并且需要和`MINIO_REGION`环境变量一致。
            ServiceURL = "http://localhost:9000", // 替换成你自己的MinIO Server的URL
            ForcePathStyle = true // 必须设为true
        });
        var amazonS3Client = new AmazonS3Client(accessKey, secretKey, config);

        // 如果你想调试与S3存储的通信的话, 可以把下一行代码取消注释, 并且实现 private void OnAmazonS3Exception(object sender, Amazon.Runtime.ExceptionEventArgs e)
    }
}
```

```
// amazonS3Client.ExceptionEvent += OnAmazonS3Exception;

var listBucketResponse = await amazonS3Client.ListBucketsAsync();

foreach (var bucket in listBucketResponse.Buckets)
{
    Console.Out.WriteLine("bucket '" + bucket.BucketName + "' created at " + bucket.CreationDate);
}
if (listBucketResponse.Buckets.Count > 0)
{
    var bucketName = listBucketResponse.Buckets[0].BucketName;

    var listObjectsResponse = await amazonS3Client.ListObjectsAsync(bucketName);

    foreach (var obj in listObjectsResponse.S3Objects)
    {
        Console.Out.WriteLine("key = '" + obj.Key + "' | size = " + obj.Size + " | tags = '" + obj.ET
        ag + "' | modified = " + obj.LastModified);
    }
}
}
```

5. 了解更多

- [AWS SDK for .NET](#)
- [配置 AWS SDK with .NET Core](#)
- [Amazon S3 for DotNet 文档](#)

如何使用aws-cli调用MinIO服务端加密

MinIO支持采用客户端提供的秘钥（SSE-C）进行S3服务端加密。客户端必须为SSE-C请求指定三个HTTP请求头：

- 算法标识符: `X-Amz-Server-Side-Encryption-Customer-Algorithm` 唯一的合法值是: `AES256`。
- 加密秘钥: `X-Amz-Server-Side-Encryption-Customer-Key` 加密秘钥必须是一个256位的base64编码的字符串。
- 加密密钥MD5校验和: `X-Amz-Server-Side-Encryption-Customer-Key-MD5` 加密密钥MD5校验和必须是秘钥的MD5和，注意是原始秘钥的MD5和，而不是base64编码之后的。

安全须知：

- 根据S3规范，minio服务器将拒绝任何通过不安全（非TLS）连接进行的SSE-C请求。这意味着SSE-C必须是TLS / HTTPS。
- SSE-C请求包含加密密钥。如果通过非TLS连接进行SSE-C请求，则必须将SSE-C加密密钥视为受损。
- 根据S3规范，SSE-C PUT操作返回的`content-md5`与上传对象的`MD5-sum`不匹配。
- MinIO Server使用防篡改加密方案来加密对象，并且不会保存加密密钥。这意味着您有责任保管好加密密钥。如果你丢失了某个对象的加密密钥，你将会丢失该对象。
- MinIO Server期望SSE-C加密密钥是高熵的。加密密钥是不是密码。如果你想使用密码，请确保使用诸如Argon2，scrypt或PBKDF2的基于密码的密钥派生函数（PBKDF）来派生高熵密钥。

1. 前提条件

从[这里](#)下载MinIO Server，并安装成带有TLS的服务。

注意一下，如果你使用的是自己签名的TLS证书，那么当你往MinIO Server上传文件时，像aws-cli或者是mc这些工具就会报错。如果你想获得一个CA结构签名的TLS证书，请参考 `Let's Encrypt`。自己签名的证书应该仅做为内部开发和测试。

2. 使用SSE-C和aws-cli

从[这里](#)下载并安装aws-cli。

假设你在本地运行了一个MinIO Server，地址是 `https://localhost:9000`，并且使用的是自己签名的证书。为了绕过TLS证书的验证，你需要指定 `--no-verify-ssl`。如果你的MinIO Server使用的是一个CA认证的证书，那你永远永远永远不要指定`--no-verify-ssl`，否则aws-cli会接受任何证书。

2.1 上传一个对象

- 创建一个名为 `my-bucket` 的存储桶：

```
aws --no-verify-ssl --endpoint-url https://localhost:9000 s3api create-bucket --bucket my-bucket
```

- 使用SSE-C上传一个对象。对象名为 `my-secret-diary`，内容来自文件 `~/my-diary.txt`。

```
aws s3api put-object \
--no-verify-ssl \
--endpoint-url https://localhost:9000 \
--bucket my-bucket --key my-secret-diary \
--sse-customer-algorithm AES256 \
--sse-customer-key MzJieXRlc2xvbmdzZWNyZXRrZXltdXN0cHJvdmlkZWQ= \
--sse-customer-key-md5 7PpPLAK26ON1VUGOWlusfg== \
--body ~/my-diary.txt
```

你需要指定你自己的加密密钥。

2.2 显示对象信息

你必须指定正确的SSE-C秘钥才能得到加密对象的元数据:

```
aws s3api head-object \
--no-verify-ssl \
--endpoint-url https://localhost:9000 \
--bucket my-bucket \
--key my-secret-diary \
--sse-customer-algorithm AES256 \
--sse-customer-key MzJieXRlc2xvbmdzzWNyZXRrZXltdXN0cHJvdmlkZWQ= \
--sse-customer-key-md5 7PpPLAK26ON1VUGOWlusfg==
```

2.3 下载一个对象

1. 删除文件 `my-diary.txt` 的本地副本:

```
rm ~/my-diary.txt
```

1. 你可以从服务器上把该文件重新下载下来:

```
aws s3api get-object \
--no-verify-ssl \
--endpoint-url https://localhost:9000 \
--bucket my-bucket \
--key my-secret-diary \
--sse-customer-algorithm AES256 \
--sse-customer-key MzJieXRlc2xvbmdzzWNyZXRrZXltdXN0cHJvdmlkZWQ= \
--sse-customer-key-md5 7PpPLAK26ON1VUGOWlusfg== \
~/my-diary.txt
```