利用MCP伺服器的強大功能：利用人工智慧驅動的解決方案徹底改變網路自動化

目錄

簡介

本文檔介紹使用行業最佳實踐構建生產就緒型模型上下文協定(MCP)伺服器的綜合參考體系結構，該體系結構通過整合Cisco Catalyst Center、ServiceNow和其他企業系統的真實實施進行了演示。MCP代表了AI系統與外部服務和資料來源互動方式的正規化轉變。但是，從原型過渡到生產需要實施企業級模式，包括身份驗證、授權、監控和可擴充性。

背景資訊

隨著組織越來越多地採用AI驅動的自動化，對穩健、安全且可擴展的整合平台的需求變得至關重要。傳統的點對點整合會產生維護開銷和安全漏洞。模型上下文協定(MCP)為AI系統整合提供了標準化的方法，但生產部署需要超越基本MCP實施的企業級功能。

本文演示了如何構建生產就緒型MCP伺服器平台，該平台包含以下內容：

1. 企業驗證:OpenID Connect(OIDC)與Cisco Duo整合
2. 精細授權:使用開放式策略代理(OPA)的Policy-as-code
3. 安全金鑰管理:用於憑證和配置的HashiCorp Vault
4. 全面監控:用於觀察和故障排除的ELK堆疊
5. 工作流協調：適用於複雜、長期運行的進程的temporal.io
6. 傳統整合:適用於現有系統的REST API代理

這為什麼重要

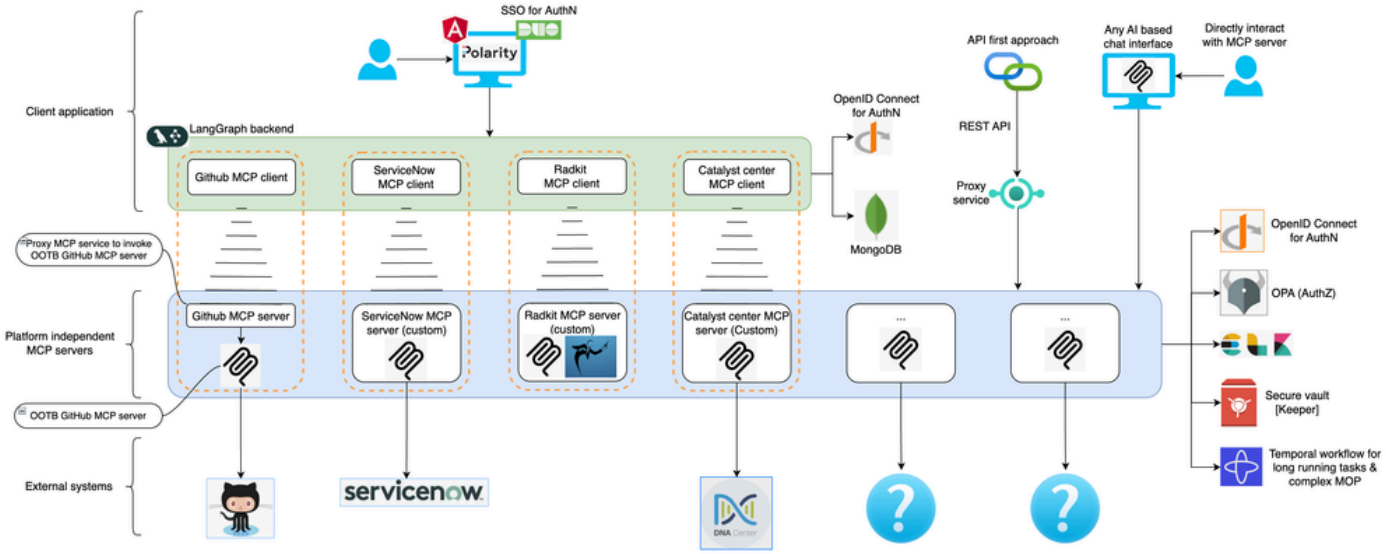
傳統整合方法存在以下幾個侷限性：

1. 安全差距:硬編碼憑證和超許可權訪問
2. 操作複雜性：難以對分散式系統進行監控和故障排除
3. 可擴充性問題:點對點整合無法隨需求的增長而擴展
4. 維護開銷:每個整合都需要自定義身份驗證和錯誤處理

採用企業模式的MCP方法可以解決這些挑戰，同時為AI驅動的自動化提供一個標準化、可重複使用的基礎。

架構概觀

參考體系結構實施了一種分層的方法，將客戶端應用程式與MCP伺服器平台分開，使多個應用程式能夠利用相同的企業級MCP基礎架構。



元件體系結構

1.客戶端應用層

客戶端層提供使用者介面和協調邏輯：

- 前端:使用Cisco Polarity UI框架的角度應用程式
- 後端:工作流協調的LangGraph多Agent系統
- 驗證:OIDC與企業身份提供商整合

2. MCP伺服器平台層

平台層通過共用服務實現企業級MCP伺服器：

核心MCP伺服器：

- `mcp-catalyst-center`:思科網路裝置管理
- `mcp-service-now`:ITSM整合和票證管理
- `mcp-github`:原始碼和儲存庫管理
- `mcp-radkit`:網路分析與監控
- `mcp-rest-api-proxy`:舊版系統整合

企業服務：

- 驗證服務:OIDC令牌驗證和使用者管理
- 授權服務:基於OPA的策略實施
- 金鑰管理:基於保管庫的憑證和配置儲存
- <Monitoring Stack:用於記錄、度量和警報的ELK
- 工作流引擎:複雜流程協調的時間性

企業安全實施

OpenID連線驗證

該平台使用OpenID Connect實施企業級身份驗證，提供與現有身份提供商的無縫整合，同時支援通過Cisco Duo進行多重身份驗證。

主要優勢

- 單一登入(SSO):使用者在所有MCP服務之間驗證一次
- 多重驗證:整合的Cisco Duo，增強安全性
- 基於令牌的安全性:使用JWT令牌的無狀態身份驗證
- 集中管理:通過現有IdP進行使用者調配和取消調配

實施概述

檔案：`mcp-common-app/src/mcp_common/oidc_auth.py`

```
"""OIDC Authentication Module - Enterprise-grade token validation with Vault integration"""

import requests
from typing import Dict, Any, Optional
from fastapi import HTTPException

def get_oidc_config_from_vault() -> Dict[str, Any]:
    """Retrieve OIDC configuration from Vault with caching."""
    vault_client = get_vault_client_with_retry()
    config = vault_client.get_secret("oidc/config")

    if not config:
        raise ValueError("OIDC configuration not found in Vault")

    # Validate required fields
    required_fields = ["issuer", "client_id", "user_info_endpoint"]
    missing_fields = [field for field in required_fields if field not in config]

    if missing_fields:
        raise ValueError(f"Missing required OIDC config fields: {missing_fields}")

    return config

def verify_token_with_oidc(token: str) -> Dict[str, Any]:
    """Verify OIDC token and extract user information."""
    config = get_oidc_config_from_vault()

    response = requests.get(
        config["user_info_endpoint"],
        headers={"Authorization": f"Bearer {token}"},
        timeout=10
    )

    if response.status_code == 200:
        user_info = response.json()
        if "sub" not in user_info:
            raise HTTPException(status_code=401, detail="Invalid token: missing subject")
        return user_info
    else:
```

```
        raise HTTPException(status_code=401, detail="Token validation failed")
```

使用開放式策略代理的細粒度授權

OPA提供靈活的策略即代碼授權，可根據使用者屬性、資源型別和上下文資訊實現細粒度訪問控制。

授權策略結構

檔案：common-services/opa/config/policy.rego

```
# Authorization Policy for MCP Server Platform - RBAC Implementation
package authz

default allow = false

# Administrative access - full permissions
allow {
    group := input.groups[_]
    group == "admin"
}

# Network engineers - Catalyst Center access
allow {
    group := input.groups[_]
    group == "network-engineers"
    input.resource == "catalyst-center"
    allowed_actions := ["read", "write", "execute"]
    allowed_actions[_] == input.action
}

# Service desk - ServiceNow and read-only network access
allow {
    group := input.groups[_]
    group == "service-desk"
    input.resource in ["servicenow", "catalyst-center"]
    input.resource == "servicenow" or input.action == "read"
}

# Developers - GitHub and REST API proxy access
allow {
    group := input.groups[_]
    group == "developers"
    input.resource in ["github", "rest-api-proxy"]
}
```

Python OPA整合

<File:mcp-common-app/src/mcp_common/opa.py

```
"""OPA Integration - Centralized authorization with audit logging"""

import os
import json
import requests
from typing import List, Dict, Any
from dataclasses import dataclass

@dataclass
class AuthorizationRequest:
    """Structure for authorization requests to OPA."""
    user_groups: List[str]
    resource: str
    action: str
    context: Dict[str, Any] = None

class OPAClient:
    """Client for interacting with Open Policy Agent (OPA) for authorization decisions."""

    def __init__(self, opa_addr: str = None):
        self.opa_addr = opa_addr or os.getenv("OPA_ADDR", "http://opa:8181")
        self.opa_url = f"{self.opa_addr}/v1/data/authz/allow"

    def check_permission(self, auth_request: AuthorizationRequest) -> bool:
        """Check if a user has permission to perform an action on a resource."""
        try:
            opa_input = {
                "input": {
                    "groups": auth_request.user_groups,
                    "resource": auth_request.resource,
                    "action": auth_request.action
                }
            }

            if auth_request.context:
                opa_input["input"]["context"] = auth_request.context

            response = requests.post(self.opa_url, json=opa_input, timeout=5)

            if response.status_code == 200:
                result = response.json()
                allowed = result.get("result", False)
                self._audit_log(auth_request, allowed)
                return allowed
            else:
                print(f"OPA authorization check failed: {response.status_code}")
                return False  # Fail secure

        except requests.RequestException as e:
            print(f"OPA connection error: {e}")
            return False  # Fail secure

    def _audit_log(self, auth_request: AuthorizationRequest, allowed: bool):
        """Log authorization decisions for audit purposes."""
        log_entry = {
            "user_groups": auth_request.user_groups,
            "resource": auth_request.resource,
            "action": auth_request.action,
            "allowed": allowed
        }
        print(f"Authorization Decision: {json.dumps(log_entry)}")
```

```
# Usage decorator for MCP server methods
def require_permission(resource: str, action: str):
    """Decorator for MCP server methods that require authorization."""
    def decorator(func):
        async def wrapper(self, *args, **kwargs):
            user_groups = getattr(self, 'user_groups', [])
            if not user_groups:
                raise Exception("User groups not found in request context")

            opa_client = OPAClient()
            auth_request = AuthorizationRequest(
                user_groups=user_groups, resource=resource, action=action
            )

            if not opa_client.check_permission(auth_request):
                raise Exception(f"Access denied for {action} on {resource}")

            return await func(self, *args, **kwargs)
        return wrapper
    return decorator
```

使用HashiCorp Vault進行安全金鑰管理

HashiCorp Vault通過加密、訪問控制和審計日誌記錄提供企業級金鑰管理。MCP平台整合了Vault，可以安全地儲存和檢索敏感資訊，包括API憑證、資料庫密碼和配置資料。

主要功能

- 靜態和傳輸中的加密:使用AES 256位加密對所有機密進行加密
- 動態機密:為外部服務生成受時間限制的憑據
- 存取控制:精細策略控制誰可以訪問哪些機密
- 稽核日誌記錄：所有秘密訪問操作的完整審計追蹤
- 金鑰輪替:憑證和證書的自動輪換

實現

檔案：`mcp-common-app/src/mcp_common/vault.py`

```
"""HashiCorp Vault Integration - Secure secret management with audit logging"""

import os
import json
import requests
from typing import Dict, Any, Optional, List
from datetime import datetime

class VaultClient:
    """Enterprise HashiCorp Vault client for secure secret management."""

    def __init__(self, vault_addr: str = None, vault_token: str = None,
                 mount_point: str = "secret"):
        self.vault_addr = vault_addr or os.getenv("VAULT_ADDR", "http://vault:8200")
        self.vault_token = vault_token or os.getenv("VAULT_TOKEN")
```

```
        self.mount_point = mount_point
        self.headers = {"X-Vault-Token": self.vault_token}

        if not self.vault_token:
            raise ValueError("Vault token must be provided or set in VAULT_TOKEN")

    def set_secret(self, path: str, secret_data: Dict[str, Any]) -> bool:
        """Store a secret in Vault KV store."""
        try:
            response = requests.post(
                f"{self.vault_addr}/v1/{self.mount_point}/data/{path}",
                headers=self.headers,
                json={"data": secret_data},
                timeout=10
            )

            success = response.status_code in [200, 204]
            self._audit_log("set_secret", path, success)
            return success

        except requests.RequestException as e:
            self._audit_log("set_secret", path, False, error=str(e))
            return False

    def get_secret(self, path: str) -> Optional[Dict[str, Any]]:
        """Retrieve a secret from Vault KV store."""
        try:
            response = requests.get(
                f"{self.vault_addr}/v1/{self.mount_point}/data/{path}",
                headers=self.headers,
                timeout=10
            )

            success = response.status_code == 200
            self._audit_log("get_secret", path, success)

            if success:
                return response.json()["data"]["data"]
            return None

        except requests.RequestException as e:
            self._audit_log("get_secret", path, False, error=str(e))
            return None

    def _audit_log(self, operation: str, path: str, success: bool, error: str = None):
        """Log secret operations for audit purposes."""
        log_entry = {
            "timestamp": datetime.utcnow().isoformat(),
            "operation": operation,
            "path": f"{self.mount_point}/{path}",
            "success": success
        }
        if error:
            log_entry["error"] = error

        print(f"Vault Audit: {json.dumps(log_entry)}")

# Usage mixin for MCP servers
class MCPSecretMixin:
    """Mixin class for MCP servers to easily access Vault secrets."""

    def __init__(self, *args, **kwargs):
```

```
        super().__init__(*args, **kwargs)
        self._vault_client = None

    @property
    def vault_client(self) -> VaultClient:
        if self._vault_client is None:
            self._vault_client = VaultClient()
        return self._vault_client

    def get_api_credentials(self, service_name: str) -> Optional[Dict[str, Any]]:
        """Get API credentials for a specific service."""
        return self.vault_client.get_secret(f"api/{service_name}")
```

核心MCP伺服器結構

每個MCP伺服器都包含與企業安全整合一致的模式：

檔案：`mcp-catalyst-center/src/main.py`

```
"""Cisco Catalyst Center MCP Server - Enterprise implementation"""

from mcp_common import VaultClient, OPAClient, get_logger, require_permission
from fastmcp import FastMCP
import os

app = FastMCP("Cisco Catalyst Center MCP Server")
logger = get_logger(__name__)

# Initialize enterprise services
vault_client = VaultClient()
opa_client = OPAClient()

@app.tool()
@require_permission("catalyst-center", "read")
async def get_all_templates(request) -> str:
    """Fetch all configuration templates from Catalyst Center."""

    # Get credentials from Vault
    credentials = vault_client.get_secret("api/catalyst-center")
    if not credentials:
        raise Exception("Catalyst Center credentials not found")

    try:
        # API call implementation
        templates = await fetch_templates_from_api(credentials)
        logger.info(f"Retrieved {len(templates)} templates")

        return {
            "templates": templates,
            "status": "success",
            "count": len(templates)
        }

    except Exception as e:
        logger.error(f"Failed to fetch templates: {e}")
        raise Exception(f"Template fetch failed: {str(e)}")
```

```
@app.tool()
@require_permission("catalyst-center", "write")
async def deploy_template(template_id: str, device_id: str) -> str:
    """Deploy configuration template to network device."""
    credentials = vault_client.get_secret("api/catalyst-center")

    # Implementation details...
    logger.info(f"Deployed template {template_id} to device {device_id}")
    return {"status": "deployed", "template_id": template_id, "device_id": device_id}
```

適用於舊版整合的REST API代理

該平台包括一個REST API代理以支援非MCP客戶端：

檔案：`mcp-rest-api-proxy/main.py`

```
"""REST API Proxy - Bridge between REST clients and MCP servers"""

from fastapi import FastAPI, HTTPException, Request
from langchain_mcp_adapters.client import MultiServerMCPClient

app = FastAPI()

# MCP server configurations
MCP_SERVERS = {
    "servicenow": "http://mcp-servicenow:8080/mcp/",
    "catalyst-center": "http://mcp-catalyst-center:8002/mcp/",
    "github": "http://mcp-github:8000/mcp/"
}

client = MultiServerMCPClient({
    server_name: {"url": url, "transport": "streamable_http"}
    for server_name, url in MCP_SERVERS.items()
})

@app.post("/api/v1/mcp/{server_name}/tools/{tool_name}")
async def execute_tool(server_name: str, tool_name: str, request: Request):
    """Execute MCP tool via REST API for legacy clients."""
    try:
        body = await request.json()

        result = await client.call_tool(
            server_name=server_name,
            tool_name=tool_name,
            arguments=body.get("arguments", {})
        )

        return {
            "status": "success",
            "result": result,
            "server": server_name,
            "tool": tool_name
        }

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Tool execution failed: {str(e)}")
```

```
@app.get("/api/v1/mcp/{server_name}/tools")
async def list_tools(server_name: str):
    """List available tools for a specific MCP server."""
    tools = await client.list_tools(server_name)
    return {"server": server_name, "tools": tools}
```

監控和可觀察性

ELK堆疊整合

該平台使用ELK堆疊實施全面的日誌記錄：

檔案：mcp-common-app/src/mcp_common/logger.py

```
"""Structured Logging for ELK Stack Integration"""

import logging
import json
from datetime import datetime
from pythonjsonlogger import jsonlogger

class StructuredLogger:
    def __init__(self, name: str, level: str = "INFO"):
        self.logger = logging.getLogger(name)
        self.logger.setLevel(getattr(logging, level.upper()))

        # JSON formatter for ELK ingestion
        formatter = jsonlogger.JsonFormatter(
            fmt='%(asctime)s %(name)s %(levelname)s %(message)s'
        )

        handler = logging.StreamHandler()
        handler.setFormatter(formatter)
        self.logger.addHandler(handler)

    def log_mcp_call(self, tool_name: str, user: str, duration: float, status: str):
        """Log MCP tool invocation with structured data."""
        self.logger.info("MCP tool executed", extra={
            "tool_name": tool_name,
            "user": user,
            "duration_ms": duration,
            "status": status,
            "service_type": "mcp_server"
        })

def get_logger(name: str) -> StructuredLogger:
    """Get configured logger instance."""
    return StructuredLogger(name)
```

關鍵監控指標

該平台跟蹤企業運營的基本指標：

- 請求延遲:每個工具的執行時間和百分比
- 身份驗證指標：成功/失敗率和響應時間
- 授權決策:政策評價頻率和成果
- Secret Access:保管庫操作和憑據使用模式
- 錯誤率:服務級別錯誤跟蹤和警報閾值
- 資源利用率:每個服務的CPU、記憶體和網路使用率

臨時工作流整合

對於複雜、長期運行的進程，該平台利用Temporal.io:

檔案：temporal-service/src/workflows/template_deployment.py

```
"""Template Deployment Workflow - Orchestrated automation with error handling"""

from temporalio import workflow, activity
from datetime import timedelta

@workflow.defn
class TemplateDeploymentWorkflow:
    @workflow.run
    async def run(self, deployment_request: dict) -> dict:
        """Orchestrate template deployment with proper error handling."""

        # Step 1: Validate template and device
        validation_result = await workflow.execute_activity(
            validate_deployment, deployment_request,
            start_to_close_timeout=timedelta(minutes=5)
        )

        if not validation_result["valid"]:
            return {"status": "failed", "reason": "Validation failed"}

        # Step 2: Create ServiceNow ticket
        ticket_result = await workflow.execute_activity(
            create_servicenow_ticket, validation_result,
            start_to_close_timeout=timedelta(minutes=2)
        )

        # Step 3: Deploy template
        deployment_result = await workflow.execute_activity(
            deploy_template, {
                **deployment_request,
                "ticket_id": ticket_result["ticket_id"]
            },
            start_to_close_timeout=timedelta(minutes=30)
        )

        # Step 4: Close ticket
        await workflow.execute_activity(
            close_servicenow_ticket, {
                "ticket_id": ticket_result["ticket_id"],
                "deployment_result": deployment_result
            },
            start_to_close_timeout=timedelta(minutes=2)
        )
```

```
        return {
            "status": "completed",
            "ticket_id": ticket_result["ticket_id"],
            "deployment_id": deployment_result["deployment_id"]
        }

@activity.defn
async def validate_deployment(request: dict) -> dict:
    """Validate deployment request against business rules."""
    # Validation logic implementation
    return {"valid": True, "validated_request": request}

@activity.defn
async def deploy_template(request: dict) -> dict:
    """Execute template deployment via Catalyst Center."""
    # Template deployment logic
    return {"deployment_id": "deploy_123", "status": "success"}
```

部署和可擴充性

容器協調

該平台使用Docker Compose進行開發。Kubernetes可用於生產：

檔案：`docker-compose.yml`(摘錄)

```
version: '3.8'
services:
  mcp-catalyst-center:
    build: ./mcp-catalyst-center
    environment:
      - VAULT_ADDR=http://vault:8200
      - OPA_ADDR=http://opa:8181
      - ELASTICSEARCH_URL=http://elasticsearch:9200
    depends_on: [vault, opa, elasticsearch]
    networks: [mcp-network]

  vault:
    image: hashicorp/vault:latest
    environment:
      VAULT_DEV_ROOT_TOKEN_ID: myroot
      VAULT_DEV_LISTEN_ADDRESS: 0.0.0.0:8200
    cap_add: [IPC_LOCK]
    networks: [mcp-network]

  opa:
    image: openpolicyagent/opa:latest-envoy
    command: ["run", "--server", "--config-file=/config/config.yaml", "/policies"]
    volumes: ["./common-services/opa/config:/policies"]
    networks: [mcp-network]
```

效能和安全注意事項

安全最佳實踐

1. 零信任架構:每個請求都經過身份驗證和授權
2. 金鑰輪替:通過Vault進行自動金鑰輪替
3. 網路分段:使用mTLS的服務網格
4. 稽核日誌記錄：ELK中的全面審計跟蹤

效能最佳化

1. 連線池:重新使用與外部API的HTTP連線
2. 快取:針對頻繁訪問的資料的基於Redis的快取
3. 非同步處理:整個堆疊的無阻塞I/O
4. 負載平衡:跨多個MCP伺服器例項分配負載

監控指標

跟蹤的關鍵指標包括：

- 每個MCP工具的請求延遲
- 身份驗證成功/失敗率
- 每個資源的授權決策
- 秘密檢索頻率
- 按服務元件統計的錯誤率

效能度量和結果

在效能測試過程中，平台實現了：

- 用於身份驗證決策的延遲低於100ms
- 所有MCP服務的99.9%正常運行時間
- 線性可擴展性多達1000個併發使用者
- 整合開發時間縮短了90%

經驗教訓和最佳做法

關鍵成功因素

1. 標準化:通用庫可減少重複和錯誤
2. 可觀察性:全面的日誌記錄可實現快速故障排除
3. 設計安全性:從第一天開始進行身份驗證和授權
4. 模組化:獨立的MCP伺服器可實現目標擴展
5. 說明文件:清除API文檔加快了採用速度

要避免的常見陷阱

1. 過度工程:開始簡單，根據需要增加複雜性
2. 緊耦合:保持MCP伺服器鬆散連線
3. 安全事後:從一開始就構建安全性
4. 測試不充分:實施全面的測試策略
5. 錯誤處理不佳:確保平穩降級

未來的增強功能

該平台規劃圖包括：

1. AI驅動的洞察:基於ML的異常檢測
2. 多雲支援:跨雲提供商部署
3. 工作流市場:可重複使用的工作流模板
4. 高級分析:即時控制板和報告

結論

構建生產級MCP伺服器需要仔細考慮企業需求，包括安全性、可擴充性、監控性和可維護性。此參考體系結構演示了如何使用行業標準工具和模式實施這些功能。

模組化設計使組織能夠逐步採用MCP，同時從第一天起就確保企業級安全性和操作。通過利用OIDC、OPA、Vault和ELK等成熟技術，團隊可以專注於業務邏輯而非基礎設施問題。

作者簡介

本文由MCP Fusioners團隊開發，作為思科內部創新計畫的一部分，展示企業AI系統整合的實用方法。

參考資料

1. 模型上下文協定規範 — https://modelcontextprotocol.io/
2. OpenID連線核心1.0 - https://openid.net/specs/openid-connect-core-1_0.html
3. 開啟策略代理文檔 — https://www.openpolicyagent.org/docs
4. HashiCorp Vault文檔 — https://www.vaultproject.io/docs
5. Temporal.io文檔 — https://docs.temporal.io/
6. ELK堆疊指南 — https://www.elastic.co/elastic-stack/

關於此翻譯

思科已使用電腦和人工技術翻譯本文件，讓全世界的使用者能夠以自己的語言理解支援內容。請注意，即使是最佳機器翻譯，也不如專業譯者翻譯的內容準確。Cisco Systems, Inc. 對這些翻譯的準確度概不負責，並建議一律查看原始英文文件（提供連結）。