譯者序

矽谷匯聚了來自世界各地的新創公司，也匯集了全球頂尖的科技人才。每年，都會有數以萬計的軟體工程師奔赴心怡的科技公司，試圖在面試中脫穎而出，成為矽谷的頂尖人才。由於招聘競爭異常激烈，求職者面對如此艱難的競爭，許多人都會感到不知所措。尤其是在面試時，求職者必需面對一堆複雜的演算法題、系統設計問題以及各類型程式碼的挑戰，心中不免產生焦慮。

作者以親身經歷和學習心得編寫出這本《矽谷頂尖 Python 工程師面試攻略》，旨在幫助每位正在準備面試的工程師，能有效率、有系統的統整自身所學，適度彰顯出自身能力，如此一來，才能在眾多求職者中脫穎而出。

本書背景雖然是設定於矽谷，但是全球各地科技公司的面試流程及面試試題皆大同小異，是以本書內容亦適用於國內、外大小科技公司。

對於希望提升自身程式編寫能力的讀者，本書的面試題，皆有詳細的解析，讓讀者能夠迅速上手，並且透過實戰演練，將理論知識轉化為實際能力，幫助讀者更深刻地理解程式流程，提升問題的解決能力，並且在面對實際工作中的挑戰時，能夠游刃有餘。

總之，這本書並非單純的考前總複習，本書是一位有經驗的導師，帶領讀者穿越面試的迷霧，逐步提升解題能力，並最終在面試中脫穎而出。如果你準備從事程式設計工作，或是想要在技術領域中走得更高、更遠，那麼這本書將是你通往成功的黃金道路。希望每位讀者都能夠從中受益，將自己打造為一位頂尖的 Python 工程師，並在職業生涯中獲取更加輝煌的成就。

資訊種子研究室

蔡文龍、何嘉益、張志成、張力元

1.1 非技術電話面試

非技術電話面試是與招聘人員快速聯繫的電話面試，通常只有 10 ～ 20 分鐘，這個過程相對簡單，沒有技術問題。招聘人員一般不是程式師，通常來自公司的人力資源部門或者「獵頭」公司。

非技術電話面試的主要目的是收集相關求職的資訊，譬如：

- 你的基本情況，包括有沒有做過某個特定專案、有沒有特定的工作技能，還有你目前的工作簽證等。
- 你需要在最近入職公司嗎？還是要在三個月內開始新工作？
- 下一份工作對你來說什麼最重要？譬如：你希望進入一個能力超強的團隊、你希望有相對自由的工作時間、你比較喜歡有趣的技術挑戰、有晉升為高級職位的空間。
- 你最感興趣的工作是什麼？是前端設計、後端設計還是機器學習？

對所有這些問題說實話，會讓招聘人員更輕鬆地獲得想要的資訊。如果招聘人員詢問你有關此工作的期望薪水，最好不要回答。說出你想知道自己和公司是否合適後，再談薪酬，會處於更好的談判位置。

1.2 技術電話面試

一般溝通之後，通常是一個或多個小時的技術電話面試。面試官會給你打電話，或告訴你透過 Skype 或 Google Handouts 加入他們的電話面試。你需要確保可以在一個網際網路連接良好的安靜地方進行面試。

面試官希望即時看到你的程式碼。這意味著要使用基於 Web 的程式碼編輯器，例如 Coderpad 或 collabedit。如果你不熟悉的話，提前在這些工具中運行一些程式碼來適應它們。

技術電話面試通常分為三個部分：

- 閒談環節（5 分鐘）。
- 技術溝通環節（30 ～ 50 分鐘）。
- 提問環節（5 ～ 10 分鐘）。

1.2.1 閒談環節

一開始的閒談不僅僅是為了幫助你放鬆，還是面試的一部分。這一環節會在 5 分鐘左右完成，面試官可能會問一些開放性問題，舉例如下：

- 簡單介紹一下自己。
- 簡單介紹一下你引以為傲的成就。
- 簡單介紹一下你簡歷裡面的項目。

在此過程中，你需要對寫在簡歷裡面的任何專案和技能都非常熟悉。

1.2.2 技術溝通環節

這是技術電話面試的核心部分，一般需要 30 ～ 50 分鐘。你可能會遇到一個較長的問題或者幾個較短的問題。

新興企業的面試官往往會問一些建構或除錯程式碼的問題。比如，編寫一個可以輸入兩個矩形並判斷它們是否重疊的函式。

較大公司的面試官將主要考查資料結構和演算法。譬如，編寫一個函式來檢查二元樹是否在 $O(n)$ 時間內是「平衡的」。他們更在乎你如何解決和優化問題。

對於這些類型的問題，最重要的是始終與面試官保持溝通。解決問題時，你將需要「大膽思考」。對於這些電話面試的技術問題，參考本書的資料結構和演算法設計部分。

CHAPTER 6

字典

Python 中的字典（Dictionary）是資料值的非順序集合，用於儲存資料值（如映射），與其他僅將單個值作為元素的資料類型不同，字典中提供了鍵值對（key-value），以使其更最佳化。

6.1 字典的基礎知識

6.1.1 建立字典

在 Python 中，可以透過將元素序列放在大括弧「{ }」中並用逗號「,」分隔來建立字典。字典包含兩個值，一個是鍵（key），另一個是鍵的值（value）。字典中的值可以是任何資料類型，並且可以重複，但鍵不能重複且必須是不變的。

注意！

字典的鍵區分大小寫，名稱若相同，但鍵的大小寫必須不同。

建立字典示例如下。

程式碼清單 6-1　建立字典

```
# 使用整數鍵建立字典
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# 使用混合鍵建立字典
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

執行結果：

```
Dictionary with the use of Integer Keys:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with the use of Mixed Keys:
{'Name': 'Geeks', 1: [1, 2, 3, 4]}
```

字典也可以透過內建函式 dict() 建立，使用大括弧「{ }」即可建立一個空字典。使用 dict() 建立字典的示例程式碼如下。

程式碼清單 6-2　利用 dict() 建立字典

```
# 建立一個空字典
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# 使用 dict() 方法建立字典
Dict = dict({1: 'Geeks', 2: 'For', 3:'Geeks'})
print("\nDictionary with the use of dict(): ")
print(Dict)

# 建立一個字典 , 每個元素成對出現
Dict = dict([(1, 'Geeks'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

執行結果：

```
Empty Dictionary:
{}

Dictionary with the use of dict():
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with each item as a pair:
{1: 'Geeks', 2: 'For'}
```

6.1.2 向字典中添加元素

在 Python 中，字典有多種方式添加元素。透過將值與鍵一起定義，使用 Dict [Key] = 'Value' 可以一次將一個鍵值對添加到字典 Dict 中。也可以使用內建的 update() 方法來更新字典中的現有值。巢狀鍵值也可以添加到現有字典中。向字典中添加元素的示例見程式碼清單 6-3。

注意！

在添加值時，如果鍵值已經存在，則該值將更新，否則將具有該值的新鍵添加到字典中。

程式碼清單 6-3　向字典中添加元素

```
# 建立一個空字典
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# 向字典中添加元素
Dict[0] = 'Geeks'
Dict[2] = 'For'
Dict[3] = 1
print("\nDictionary after adding 3 elements: ")
print(Dict)

# 向字典中添加一組元素
Dict['Value_set'] = 2, 3, 4
print("\nDictionary after adding 3 elements: ")
```

```
print(Dict)

# 更新字典中的鍵值
Dict[2] = 'Welcome'
print("\nUpdated key value: ")
print(Dict)

# 添加巢狀的字典到字典的鍵值
Dict[5] = {'Nested' :{'1' : 'Life', '2' : 'Geeks'}}
print("\nAdding a Nested Key: ")
print(Dict)
```

執行結果：

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Geeks', 2: 'For', 3: 1}

Dictionary after adding 3 elements:
{0: 'Geeks', 2: 'For', 3: 1, 'Value_set': (2, 3, 4)}

Updated key value:
{0: 'Geeks', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4)}

Adding a Nested Key:
{0: 'Geeks', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4), 5: {'Nested': {'1':
     'Life', '2': 'Geeks'}}}
```

6.1.3 存取字典中的元素

可以透過鍵名存取字典中的元素，示例程式碼如下。

程式碼清單 6-4　存取字典中的元素

```
# 從字典中存取元素
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using key
print("Accessing a element using key:")
print(Dict['name'])

# accessing a element using key
```

CHAPTER 8

鏈結串列

鏈結串列（Linked list，或稱鏈表）與陣列相似，鏈結串列也是一種線性資料結構。如圖 8-1 所示，鏈結串列中的每個元素實際上都是一個單獨的物件，而所有物件都透過每個元素中的引用欄位連結在一起。鏈結串列有兩種型別：單鏈結串列和雙鏈結串列。圖 8-1 是一個單鏈結串列，圖 8-2 是一個雙鏈結串列。

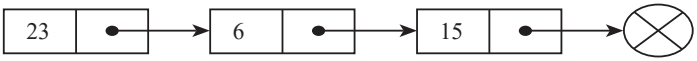

圖 8-1　單鏈結串列

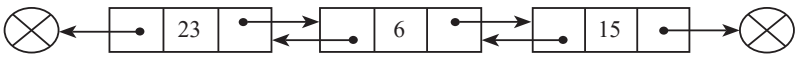

圖 8-2　雙鏈結串列

8.1 雙指標技術

有兩種使用雙指標技術的方案。

- 兩個指標從不同的位置開始：一個指標從頭開始，另一個指標從尾開始。
- 兩個指標以不同的速度移動：一個指標較快，而另一個指標可能較慢。

對於單鏈結串列，由於只能在一個方向上走訪鏈結串列，因此第一種方案不起作用，而第二種方案（也稱為慢指標和快指標技術）非常有用。本章將重點介紹如何使用鏈結串列中的慢指標和快指標技術解決問題。

8.2 實例 1：判斷鏈結串列是否有循環

指定一個鏈結串列，如何確定其是否有循環？為了表示指定鏈結串列中的循環，使用整數 pos 來表示尾部連接到鏈結串列中的位置（0 索引）。如果 pos 為 -1，則鏈結串列中沒有循環。舉例如下。

- 輸入：head = [3, 2, 0, -4]，pos = 1
- 輸出：True

說明：鏈結串列中有一個循環，尾巴連接到第二個節點，如圖 8-3 所示。

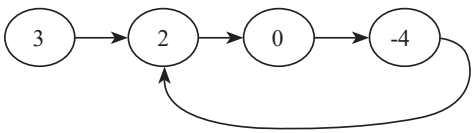

圖 8-3　循環鏈結串列

解題思路：利用兩個指標，一快一慢，如果相遇就說明有循環。示例程式碼如下。

程式碼清單 8-1　判斷鏈結串列是否有循環

```
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        slow = head
        fast = head

        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if fast:
                if fast.val == slow.val:
                    return True
        return False
```

8.3　實例 2：兩個鏈結串列的交集

編寫程式以搜尋兩個單鏈結串列的交點開始的節點。如圖 8-4 所示，以下兩個鏈結串列，交點開始的節點是 $c1$。

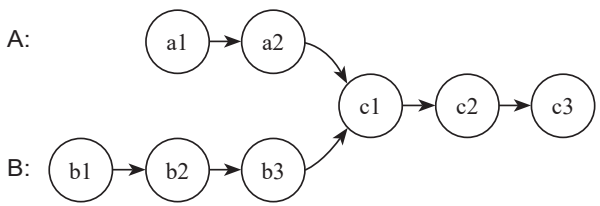

圖 8-4　兩個鏈結串列的交集

解題思路：可以利用雜湊表，把其中一個鏈結串列的所有節點保存下來。然後走訪另外一個鏈結串列，如果在雜湊表中找到相同的節點，則傳回。示例程式碼如下。

程式碼清單 8-2　利用雜湊表尋找鏈結串列的交集

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        a=set()
        while(headA):
            a.add(headA)
            headA=headA.next
```

```
        while(headB):
            if(headB in a):
                return(headB)
            headB=headB.next
        return(None)
```

當然，還有一個解題思路，就是雙指標操作，確保兩個鏈結串列具有相同的長度，然後走訪，尋找是否具有相同節點。示例程式碼如下。

程式碼清單 8-3　利用雙指標尋找鏈結串列的交集

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        def size(head: ListNode):
            n = 0
            while head:
                n += 1
                head = head.next
            return n
        # 獲得鏈結串列的長度
        nA = size(headA)
        nB = size(headB)

        if nA==0 or nB==0:
            return

        itr1 = headA
        itr2 = headB

        # 找出鏈結串列長度之差
        d = nA-nB
        if d>0:
            while d>0:
                itr1 = itr1.next
                d = d -1
        else:
            while d<0:
                itr2 = itr2.next
                d = d+1
        # 目前兩個鏈結串列長度一致 , 開始走訪
        while itr1 != itr2:
            itr1 = itr1.next
            itr2 = itr2.next
        return itr1
```

8.4 實例 3：複製隨機鏈結串列

與上面尋找鏈結串列的交集比較接近的，就是複製一個鏈結串列。

指定一個鏈結串列，使得每個節點都包含一個額外的隨機指標，該指標可以指向鏈結串列中的任何節點或為空（null），傳回鏈結串列的深層拷貝。

鏈結串列在輸入 / 輸出中表示為 n 個節點的串列。每個節點表示為一對 [val，random_ index]，其中：val 表示 Node.val 的整數；random_index 表示隨機指標指向節點的索引（範圍從 0 到 n - 1），如果不指向任何節點，則為 null。

解題思路：主要是走訪鏈結串列裡面的每個 next 節點和 random 節點。當然需要利用雜湊表來儲存已經複製的節點，防止多次生成。示例程式碼如下。

程式碼清單 8-4　複製隨機鏈結串列

```
class Solution:
    def copyRandomList(self, head: 'Node') -> 'Node':
        if not head: return None
        table = {}
        table[head] = Node(head.val)
        curr = head

        while curr:
            copy = table[curr]
            if curr.next is not None:
                if curr.next not in table:
                    copy.next = Node(curr.next.val)
                    table[curr.next] = copy.next
                else:
                    copy.next = table[curr.next]
            if curr.random is not None:
                if curr.random not in table:
                    copy.random = Node(curr.random.val)
                    table[curr.random] = copy.random
                else:
                    copy.random = table[curr.random]
                    copy = copy.next
                    curr = curr.next
                return table[head]
```

CHAPTER 20

系統設計理論

一般系統設計主要是考查物件導向的設計，或是巨量資料（big data，或稱大數據）系統分析。針對系統設計的面試題往往是一個開放式的對話，面試官會期望應聘人主導這個對話。

20.1 設計步驟

以巨量資料系統為例，可以透過下面的步驟來完成系統設計的面試題。

20.1.1 描述使用場景、約束和假設

把所有必要的資訊整理在一起，來審視面試題，持續提問以明確系統使用場景和約束，並討論假設條件。思考誰會使用這個系統？他們會怎樣使用它？有多少用戶？系統的主要功能是什麼？系統的輸入、輸出分別是什麼？期望它能處理多少資料，每秒處理多少個請求，以及讀寫比率是多少？

20.1.2 建立策略規劃

巨量資料系統的策略規劃是一個綜合規劃，涉及資料蒐集、儲存、處理、分析、應用和安全等方面。以下是巨量資料系統策略規劃的關鍵要素。

1. **需求分析**：首先，瞭解業務需求和目標。明確要解決的問題、資料的來源、資料類型和資料量，以及所需的分析和報告。

2. **資料採集與收集**：確定資料的採集和收集策略。考慮如何從不同來源（例如資料庫、紀錄、感測器、社群媒體）收集資料，並確保資料的高品質和一致性。

3. **資料儲存**：選擇適當的資料儲存方案。這可能包括分散式檔案系統（如 HDFS）、欄式資料庫（Columnar Database）、資料湖（data lake）、主記憶體資料庫和 NoSQL（No Only SQL）資料庫。資料儲存層應該支援擴展性、容錯性和高可用性。

4. **資料處理**：設計資料處理層面，包括批次處理引擎、資料流處理引擎和查詢引擎。使用適當的工具和技術來處理與轉換資料，以滿足分析和應用的需求。

5. **資料分析和探勘**：定義資料分析和探勘（mining，或稱挖掘）任務，包括資料淨化（cleaning，或稱清洗）、轉換、模型訓練和預測等。使用機器學習和資料探勘演算法來獲得有價值的資訊。

6. **資料應用和視覺化**：開發資料應用程式和視覺化工具，以將分析結果傳遞給最終用戶，例如儀錶板、資料報告和決策支援系統。

7. **資料安全性和合規性**：確保資料的安全性和合規性，包括資料加密、身分驗證、訪問控制和合規性檢查。

8. **性能和可伸縮性**：考慮系統的性能和可伸縮性。使用負載平衡和快取來提高性能，並實施彈性伸縮策略以適應負載變化。

9. **監控和管理**：實施監控工具來追蹤系統性能、資源利用率和異常。建立日誌紀錄和審計機制以支援故障排除和合規性。

10. **備份和容錯**：開發備份和容錯策略，以確保資料的安全和可復原性。

11. **定期評估和改進**：進行定期的系統評估和性能測試，以保持系統的高效運行，並根據需求進行適時改進和最佳化。

12. **培訓和支援**：建立培訓計畫，確保團隊熟悉巨量資料系統的運維工作，並提供支援以解決問題和應對挑戰。

這些要素是巨量資料系統策略規劃的基礎，能夠確保系統滿足業務需求、高效運行並持續演進。設計應根據組織的需求和資源進行訂製，以便建立出一個強大、高性能的巨量資料系統。面試的時候一般需要把給定的問題轉化成系統策略規劃。

20.1.3 設計核心元件

巨量資料系統的核心元件是建立和管理大規模資料的關鍵部分。對每一個核心元件進行深入的分析。以下是巨量資料系統中的一些核心元件。

1. **分散式檔案系統**（Distributed File System，DFS，或稱分布式文件系統）：這是巨量資料系統的核心元件之一，用於儲存大量分佈在多個節點上的資料。一些常見的 DFS 包括 HDFS 和 Amazon S3。

2. **批次處理引擎**： 批次處理引擎（或稱批處理引擎）用於處理大規模批量資料。Hadoop MapReduce 和 Apache Spark 是常見的批次處理引擎，它們支援分散式運算。

3. **串流處理引擎**：串流處理引擎（或稱流處理引擎）用於處理即時資料流，允許對資料進行即時分析和回應。Apache Kafka 和 Apache Flink 是串流處理引擎的示例。

4. **查詢和分析引擎**：查詢和分析引擎允許使用者查詢和分析儲存在巨量資料系統中的資料，常見的工具有 Apache Hive、Presto、Apache Impala 和 Apache Drill。

5. **資料倉儲**：資料倉儲（data warehouse，或稱數據倉庫）是一種專門用於儲存和查詢資料的資料庫系統，通常用於支持商業智慧（BI）和資料分析。常用的資料倉儲包括 Amazon Redshift、Google BigQuery 和 Snowflake。

6. **機器學習和深度學習框架**：巨量資料系統通常包括機器學習和深度學習框架，用於建立和訓練模型，以進行預測和分類。TensorFlow、PyTorch、Scikit-learn 等是常用的機器學習和深度學習框架。

7. **NoSQL 資料庫**： NoSQL 資料庫用於儲存半結構化和非結構化資料，以支援巨量資料應用程式。MongoDB、Cassandra、HBase 和 Couchbase 是常見的 NoSQL 資料庫。

8. **快取層**：快取層用於加速資料訪問、降低儲存和計算的負載。快取層工具包括 Redis、Memcached 和 Apache Kafka。

9. **資料管理工具和中繼資料儲存**：資料管理工具和元資料（或稱元數據、中繼資料）儲存用於管理資料和追蹤資料的來源、定義和品質。

10. **資料視覺化工具**：資料視覺化工具用於將分析結果視覺化，以便用戶理解資料並制定決策。Tableau、Power BI 和 Matplotlib 等是常見的資料視覺化工具。

11. **監控和日誌記錄工具**：監控工具用於追蹤系統性能和資源利用率，而日誌記錄工具用於記錄操作紀錄和審計資料。

12. **負載平衡器**：負載平衡器（load balancer，或稱負載均衡器）用於平衡請求和任務，確保資源有效分配。

這些核心元件是建立巨量資料系統的關鍵部分，根據組織的需求和具體的巨量資料用例，可以選擇和配置適當的元件。巨量資料系統通常採用分散式和雲端架構，以應對大規模資料的挑戰，同時保持高性能、可擴展性和可靠性。

例如，如果你被問到「**設計一個 *URL* 縮寫服務**」，則可以討論以下要點。

1. 生成並儲存一個完整 URL 的雜湊值。

   使用 MD5 還是 Base62：我們可以使用 MD5 和 Base62 這兩種演算法來獲得隨機雜湊值（hash value，或稱哈希值）。實際上兩種演算法均可，但本次我們使用 Base62，因為該演算法可以生成超過 3 萬億個字串組合。而 MD5 雜湊有一個小問題，是它會給出 20 ~ 22 個字元長的雜湊值，而我們只需要 7 個字元。

   雜湊碰撞（collision，或稱哈希碰撞）是指在雜湊函式中，兩個不同的輸入值（通常是資料或關鍵字）對應到相同的雜湊值的情況。雜湊函式的目標是將資料均勻分散到雜湊表或雜湊桶中，以便在進行搜尋、插入和刪除操作時能夠快速訪問資料。然而，由於輸入空間遠大於雜湊值的輸出空間，雜湊碰撞是不可避免的。

   使用 SQL 還是 NoSQL：根據經驗，如果需要分析資料的行為或建立自訂儀錶板，則使用 RDBMS（關聯式資料庫管理系統）及 SQL 是更好的選擇。此外，SQL 通常允許更快的資料儲存和復原，並且可以處理更複雜的查詢。如果想擴展 RDBMS 的標準結構或者需要建立靈活的架構，則 NoSQL 將是更好的選擇。

   資料庫模型。資料庫模型描述在資料庫中結構化和操縱資料的方法。模型的結構部分規定資料如何被描述（例如樹、資料表等）。模型的操作部分規定資料的增加、刪除、顯示、維護、列印、搜尋、選擇、排序和更新等操作。

2. 將一個 hashed URL 轉成完整的 URL。

3. 討論如何進行物件導向設計以及 API 的設計。

20.1.4 擴展設計

這裡結合圖 20-1 來簡單介紹巨量資料架構設計的基本知識點。

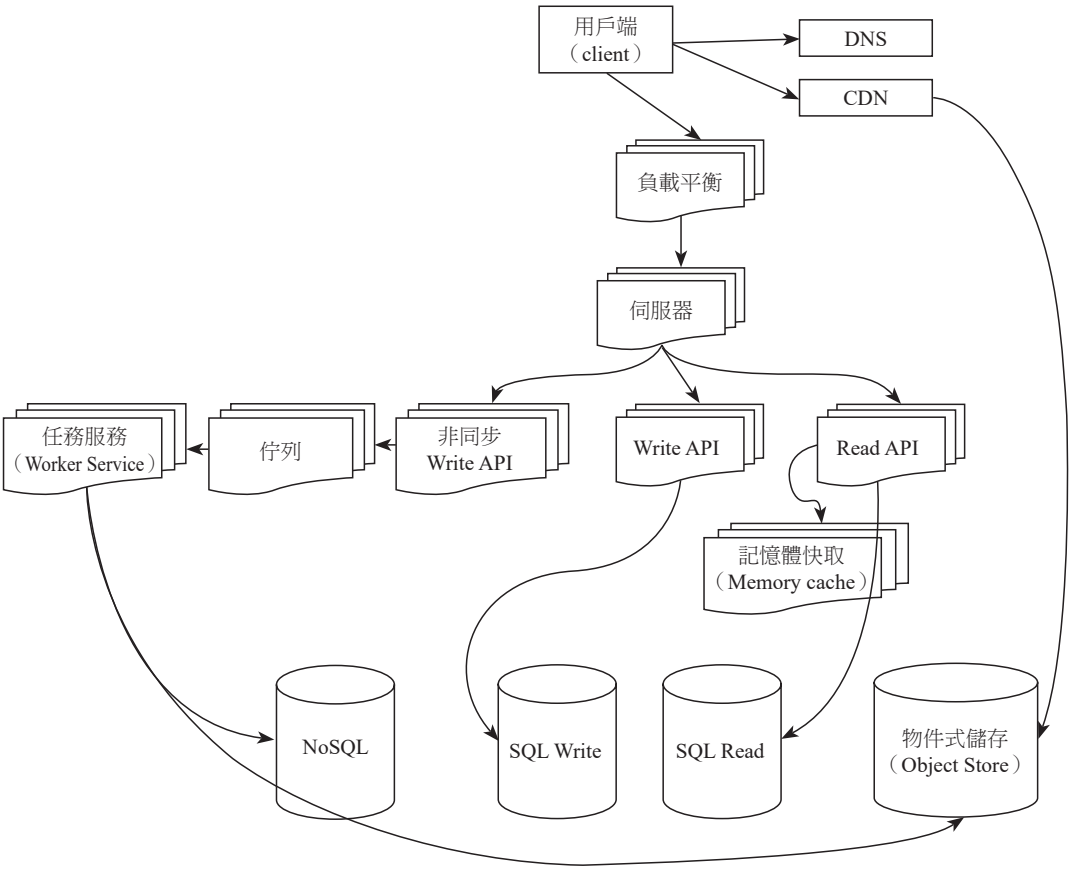

圖 20-1 巨量資料架構設計圖

巨量資料架構的擴展設計，是為了應對不斷增長的資料量和更高的性能需求。以下是關於巨量資料架構擴展設計的一些建議：

1. **分散式儲存的擴展**：如果儲存層使用的是分散式檔案系統（如 HDFS）或雲端儲存（如 Amazon S3），可以考慮增加更多的儲存節點，以增加容量。此外，確保儲存層可以水平擴展，以應對資料量的增長。

2. **計算資源的擴展**：在資料處理層面，如批次處理引擎和串流處理引擎，考慮增加更多的計算節點，以提高處理能力。可以使用雲端服務提供者的彈性計算資源，根據需求動態擴展或縮小資源。

3. **資料分區**：將資料分成更小的分區，以便進行並行處理和負載平衡。這可以提高性能，並縮短單個任務或查詢的執行時間。

4. **資料壓縮和歸檔**：針對歷史資料，可以實施資料壓縮和歸檔策略，以降低儲存成本，但仍然能夠快速檢索和分析舊資料。

5. **快取層的增強**：在資料處理層增加快取，以減輕儲存和計算資源的負載，提高回應速度。

6. **負載平衡**：使用負載平衡器來將請求和任務均勻分佈到不同的計算節點和資料節點，以確保資源有效利用。

7. **資料分區策略**：考慮採用更精細的資料分區策略，以使資料在不同節點之間更均勻地分佈，來防止熱點。

8. **自動化和彈性伸縮**：實施自動化的資源調整和彈性伸縮策略，以根據負載動態分配和釋放資源。

9. **備份和災難復原**：實施有效的備份和災難復原（Elastic Disaster Recovery，或稱容災）計畫，以防止資料丟失和系統中斷。在不同的地理位置備份資料。

10. **安全性和合規性**：隨著資料量的增加，確保資料的安全性和合規性變得更加重要。加強資料加密、身分驗證和審計，以滿足合規性要求。

11. **監控和性能調整**：使用監控工具來追蹤系統性能和資源使用情況，及時識別問題並進行性能調整。

12. **定期評估和規劃**：定期評估巨量資料架構的性能和需求，制定長期規劃，以應對未來的需求。

巨量資料架構的擴展設計是一個持續的過程，需要根據不斷變化的需求和技術發展進行調整。透過合理規劃和實施擴展策略，可以確保巨量資料平臺高效地處理大規模資料，並滿足組織的需求。

20.2 網域名稱系統

網域名稱系統（Domain Name System，DNS）是網際網路的核心組成部分，它用於將人類可讀的網域名稱（簡稱域名）如 www.example.com 轉換為電腦可理解的 IP 位址（IP address，或稱 IP 地址）如 192.0.2.1，其工作原理如圖 20-2 所示。

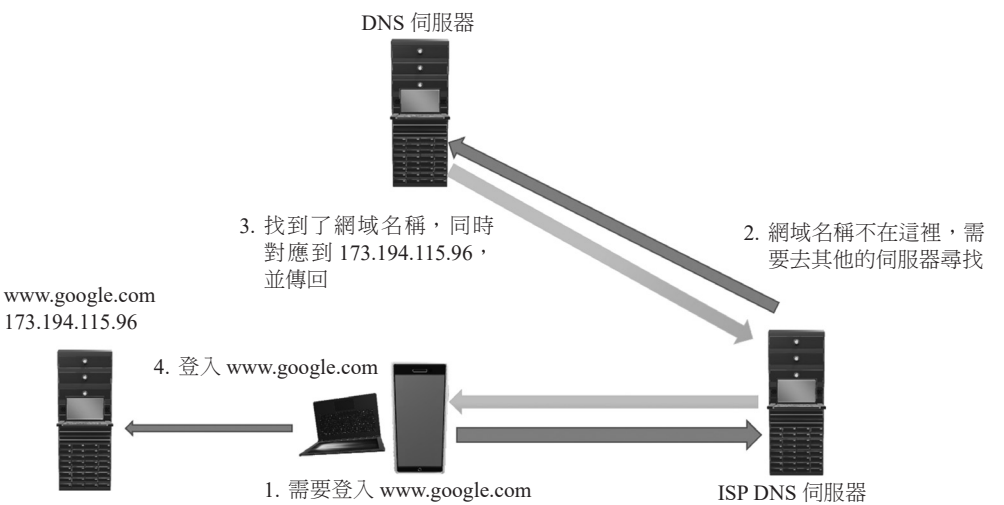

圖 20-2 網域名稱系統的工作原理

DNS 的工作原理如下。

1. **網域名稱查詢請求**：在瀏覽器中輸入一個網址（例如 www.example.com），瀏覽器首先會檢查本地 DNS 快取（本地主機檔）以搜尋對應的 IP 位址。如果沒有找到，瀏覽器將向本地 DNS 伺服器發出查詢請求。

2. **本地 DNS 伺服器查詢**：本地 DNS 伺服器是由網際網路服務提供者（ISP）或所連接的網路（如公司或學校）所提供。本地 DNS 伺服器會嘗試在其快取中搜尋相應的 IP 位址。如果找到了，它將直接傳回 IP 位址給瀏覽器。

3. **遞迴查詢**：如果本地 DNS 伺服器沒有所需網域名稱的 IP 位址，它會執行遞迴查詢。在遞迴查詢中，本地 DNS 伺服器將向根網域名稱伺服器發送請求，請求根網域名稱伺服器提供頂層網域名稱伺服器（TLD 伺服器，例如 .com）的 IP 位址。

4. **根網域名稱伺服器**：根網域名稱伺服器是 DNS 階層的最高層，管理頂層網域名稱伺服器的資訊。根網域名稱伺服器返回本地 DNS 伺服器所需頂層網域名稱伺服器的 IP 位址。

5. **頂層網域名稱伺服器**：本地 DNS 伺服器接下來會向頂層網域名稱伺服器發出請求，請求頂層網域名稱伺服器提供二級網域名稱伺服器的 IP 位址。例如，對於 .com 網域，頂層網域名稱伺服器將返回 example.com 網域名稱伺服器的 IP 位址。

6. **網域名稱伺服器**：本地 DNS 伺服器獲得了二級網域名稱伺服器的 IP 位址後，繼續向該網域名稱伺服器發出請求。網域名稱伺服器通常由網站的託管提供商管理，並儲存有關特定網域名稱的 DNS 紀錄。

7. **DNS 紀錄查詢**：網域名稱伺服器根據請求回傳相應的 DNS 紀錄，例如，A 紀錄（將網域名稱對應到 IPv4 位址）或 AAAA 紀錄（將網域名稱對應到 IPv6 位址）。

8. **本地 DNS 伺服器快取**：本地 DNS 伺服器將從網域名稱伺服器接收的 DNS 紀錄儲存在其快取中，以便將來的查詢使用，這可以減少對外部 DNS 伺服器的依賴。

9. **返回 IP 位址**：最終，本地 DNS 伺服器將所需的 IP 位址回傳給瀏覽器，瀏覽器將使用該 IP 位址建立與目標伺服器的連接。

整個過程是逐級查詢，從根網域名稱伺服器到頂層網域名稱伺服器，再到網域名稱伺服器，最終到達目標網域名稱的 DNS 紀錄。這使得 DNS 能夠有效地將網域名稱轉換為 IP 位址，從而使網際網路用戶能夠輕鬆地訪問各種網站和資源。

20.3 負載平衡器

負載平衡器將傳入的請求分發到應用伺服器和資料庫等計算資源。無論哪種情況，負載平衡器都將來自計算資源的回應回傳給恰當的用戶端。

負載平衡器的效用在於：防止請求進入不好的伺服器、防止資源超載、幫助消除單一的故障點。

負載平衡器的設計如圖 20-3 所示。

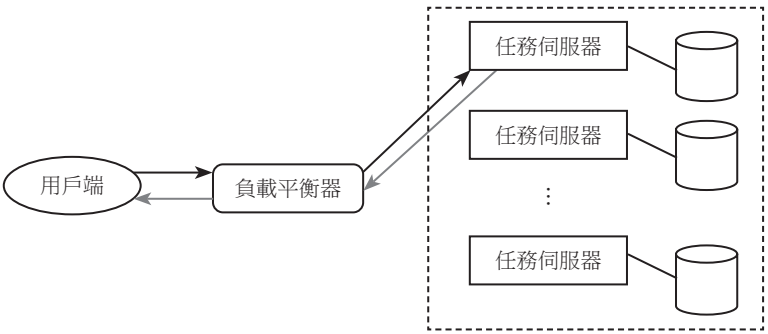

圖 20-3　負載平衡器的設計

從圖 20-3 中可以看到，使用者訪問負載平衡器，再由負載平衡器將請求轉發給後端伺服器。在這種情況下，單點故障移轉到負載平衡器上了，又可以透過引入第二個負載平衡器來緩解。但在討論之前，我們先探討負載平衡器的工作方式。

負載平衡演算法是用於在多個伺服器之間分配負載（如網路請求、資料流量等）的策略，以確保伺服器資源的有效利用，提高性能和可用性。以下是一些常見的負載平衡演算法。

1. **輪詢**（Round Robin）：這是最簡單的負載平衡演算法之一，它按照順序將每個請求分配給下一個伺服器。當到達伺服器串列的末尾時，輪詢重新開始。輪詢對於具有相似硬體性能的伺服器非常有效。

2. **加權輪詢**（Weighted Round Robin）：在加權輪詢中，每個伺服器都被賦予一個權重，以決定每個伺服器獲得多少請求。這對於伺服器性能不均等情況非常有用。

3. **最少連接**（Least Connections）：這個演算法會將請求分配給當前連接數最少的伺服器。它適用於伺服器之間負載不均等的情況。

4. **加權最少連接**（Weighted Least Connections）：類似於最少連接演算法，但會考慮每個伺服器的權重，以便更好地處理性能不均等的情況。

5. IP 雜湊（IP Hash）：這種方法使用用戶端的 IP 位址來計算雜湊值，將請求分配給特定的伺服器。這對於確保相同用戶端的請求都被發送到相同的伺服器時很有用。

6. URL **雜湊**（URL Hash）：這種方法使用請求的 URL 來計算雜湊值，以決定請求應該發送到哪個伺服器。這對於快取和內容傳遞網路（CDN，或稱內容分發網路）等應用很有用。

7. **最短回應時間**（Least Response Time，或稱最少響應時間）：這個演算法會將請求發送到回應時間最短的伺服器。這需要即時監控伺服器的回應時間。

8. **隨機**（Random）：隨機演算法隨機選擇一個伺服器來處理請求。儘管它非常簡單，但在某些情況下卻很有效。

9. **來源 IP 雜湊**（Source IP Hash，或稱源 IP 雜湊）：這種方法使用用戶端的來源 IP 位址來計算雜湊值，以決定請求應該發送到哪個伺服器。它適用於會話保持的應用。

10. **最大連接**（Maximum Connections）：這個演算法將請求分配給允許的最大連接數未達到上限的伺服器。

不同的負載平衡演算法適用於不同的應用場景，通常根據伺服器性能、應用需求和負載分佈等因素選擇合適的演算法。一些負載平衡器還支援自訂演算法，以滿足特定需求。

20.4 分散式快取系統

Memcached 是一個高性能的分散式快取（distributed cache，或稱分布式緩存）系統，廣泛用於加速應用程式的資料訪問。它採用了分散式架構，允許將資料儲存在多個伺服器節點上，並透過雜湊演算法將資料分佈到這些節點上，以提高性能和可伸縮性。

Memcached 一般透過記憶體資料庫查詢結果，可減少資料庫的訪問次數，以提高動態 Web 應用的速度和擴展性。分散式快取的架構如圖 20-4 所示。

在快取儲存容量已滿的情況下，刪除快取物件需要考慮多種機制：一種是按佇列機制；另一種是根據快取物件的優先順序。Memcached 會優先使用已超時的紀錄空間，但即使如此，也會發生追加新紀錄時空間不足的情況。此時就要使用名為最近最少使用（LeastRecently Used，LRU）的機制來分配空間。因此當 Memcached 的記憶體空間不足並且獲取到新紀錄時，就在最近未使用的紀錄中搜尋，將該空間分配給新的紀錄。

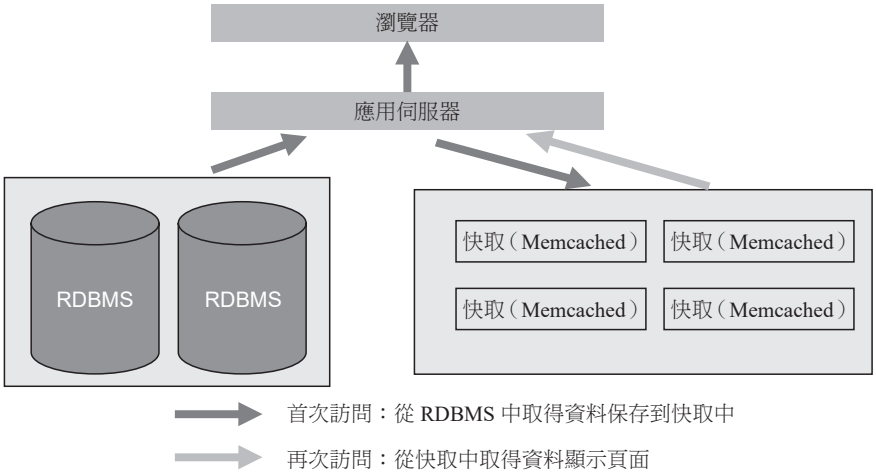

圖 20-4　分散式快取的架構

Memcached 雖然稱為「分散式」快取伺服器，但伺服器端並沒有分散式的功能。Memcached 的分散式完全是由用戶端實現，下面舉例說明 Memcached 是如何實現分散式快取。

例如，假設有三台 Memcached 伺服器 Node1 ～ Node3，應用程式要保存鍵名為“tokyo”、“kanagawa”、“chiba”、“saitama”、“gunma”等資料。

首先在 Memcached 中增加“tokyo”。將“tokyo”傳給用戶端程式庫後，用戶端實現的演算法，就會根據「鍵」來決定保存資料的 Memcached 伺服器。伺服器選取後，即命令它保存鍵“tokyo”及其值。同樣，鍵“kanagawa”、“chiba”、“saitama”、“gunm”也都是先選擇伺服器再保存。

接下如何來讀取保存的資料，這時要將要讀取的鍵“tokyo”傳遞給函式庫。函式庫利用與資料保存時相同的演算法，根據「鍵」選擇伺服器，然後發送 get 命令。只要資料沒有因某些原因被刪除，就能取得保存的值。

Memcached 伺服器實現分散式快取的原理如圖 20-5 所示。

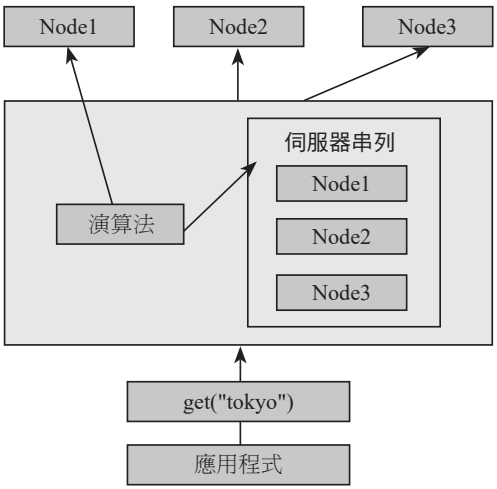

圖 20-5　Memcached 伺服器實現分散式快取的原理

這樣，將不同的鍵保存到不同的伺服器上，就實現了分散式快取。Memcached 伺服器增多後，鍵就會分散，即使一台 Memcached 伺服器發生故障無法連接，也不會影響其他快取，系統依然能繼續運行。

以下是關於 Memcached 分散式系統的一些要點。

1. **節點和叢集**：Memcached 分散式系統由多個節點組成，每個節點可以是一台獨立的伺服器。這些節點一起構成了 Memcached 叢集（Cluster，或稱集群），負責儲存快取資料。

2. **資料分片**：Memcached 使用分片（Sharding）策略，將資料分成多個片段。每個節點負責儲存其中的一個或多個資料片段。

3. **一致性雜湊**：為了確定哪個節點儲存特定的資料，Memcached 使用了一致性雜湊（Consistent Hashing）演算法。這個演算法利用雜湊函式將資料的鍵對應到環形空間，然後選擇離雜湊值最近的節點來儲存資料。

4. **節點的動態加入和退出**：Memcached 支援節點的動態加入和退出。當節點加入或退出叢集時，一致性雜湊演算法會重新分佈資料，確保資料均勻分佈，並避免大規模資料移轉。