

程式人



數學、程式和機器

(從數理邏輯、 λ -Calculus、圖靈機到現代電腦的那段歷史)

陳鍾誠

2022年4月20日

第一個故事

數學

故事的起源

- 得從數學家希爾伯特 (David Hilbert)
開始說起

希爾伯特

大衛·希爾伯特
David Hilbert



大衛·希爾伯特攝於1886年

1900 年

- 希爾伯特在一場德文演講中提出了 20 世紀希望能解決的 23 個數學問題

希爾伯特的23個問題 [編輯]

維基百科，自由的百科全書

希爾伯特問題（德語：Hilbertsche Probleme）是德國數學家大衛·希爾伯特於1900年在巴黎舉行的第二屆國際數學家大會上作了題為《數學問題》的演講，所提出23道最重要的數學問題。希爾伯特問題對推動20世紀數學的發展起了積極的推動作用。在許多數學家努力下，希爾伯特問題中的大多數在20世紀中得到了解決。

希爾伯特問題中未能包括拓撲學、微分幾何等領域，除數學物理外很少涉及應用數學，更不曾預料到電腦的發展將對數學產生重大影響。20世紀數學的發展實際上遠遠超出了希爾伯特所預示的範圍。

希爾伯特問題中的1-6是數學基礎問題，7-12是數論問題，13-18屬於代數和幾何問題，19-23屬於數學分析。

這 23 個問題

- 影響非常深遠！

#	主旨	進展	說明
第1題	連續統假設	部分解決	1963年美國數學家保羅·柯恩以力迫法證明連續統假設不能由策梅洛-弗蘭克爾集合論（無論是否含選擇公理）推導。也就是說，連續統假設成立與否無法由ZF/ZFC確定。
第2題	算術公理之相容性	部分解決	庫爾特·哥德爾在1931年證明了哥德爾不完備定理，但此定理是否已回答了希爾伯特的原始問題，數學界沒有共識。
第3題	兩四面體有相同體積之證明法	已解決	答案：否。1900年，希爾伯特的學生馬克斯·登以一反例證明了是不可以的。
第4題	建立所有度量空間使得所有線段為測地線	太隱晦	希爾伯特對於這個問題的定義過於含糊。
第5題	所有連續群是否皆為可微群	已解決	1953年日本數學家山邊英彥證明在無「小的子群」情況下，答案是肯定的 ^[1] ；但此定理是否已回答了希爾伯特的原始問題，數學界仍有爭論。
第6題	公理化物理	部分解決	希爾伯特後來對這個問題進一步解釋，而他自己也進一步研究這個問題。科摩哥洛夫對此也有貢獻。然而，儘管公理化已經開始滲透到物理當中，量子力學中仍有至今不能邏輯自治的部分（如量子場論），故該問題未完全解決。
第7題	若 b 是無理數、 a 是除0、1之外的代數數，那麼 a^b 是否超越數	已解決	答案：是。分別於1934年、1935年由蘇聯數學家亞歷山大·格爾豐德與德國數學家特奧多爾·施耐德獨立地解決。

目前，有些問題已經解決

第8題	黎曼猜想及哥德巴赫猜想和 孿生質數猜想	未解決	雖然分別有比較重要的突破和被解決的弱化情況，三個問題均仍未被解決。
第9題	任意代數數體的一般互反律	部分 解決	1927年德國的 埃米爾·阿廷 證明在阿貝爾擴張的情況下答案是肯定的；此外的情況則尚未證明。
第10題	不定方程式可解性	已解 決	答案：否。1970年由蘇聯數學家 尤里·馬季亞謝維奇 證明。
第11題	代數係數之二次形式	部分 解決	有理數的部分由哈塞於1923年解決。
第12題	一般代數數體的阿貝爾擴張	未解 決	埃里希·赫克於1912年用希爾伯特模形式研究了實二次體的情形。虛二次體的情形用複乘理論已基本解決。一般情況下則尚未解決。
第13題	以二元函數解任意七次方程 式	部分 解決	1957年蘇聯數學家科摩哥洛夫和弗拉基米爾·阿諾爾德證明對於單值解析函數，答案是否定的；然而希爾伯特原本可能希望證明的是代數函數的情形，因此該問題未獲得完全解答。
第14題	證明一些函數完全系統 (Complete system of functions) 之有限性	已解 決	答案：否。1962年日本人 永田雅宜 提出反例。
第15題	舒伯特演算之嚴格基礎	部分 解決	一部分在1938年由范德瓦爾登得到嚴謹的證明。

有些問題還在研究中

第16題	代數曲線及表面之拓撲結構	未解決	此問題進展緩慢，即使對於度為8的代數曲線也沒有證明。
第17題	把有理函數寫成平方和分式	已解決	答案：是。1927年 <u>埃米爾·阿廷</u> 解決此問題，並提出實封閉體。 [2] [3]
第18題	非正多面體能否密鋪空間、球體最緊密的排列	已解決	1911年 <u>比伯巴赫</u> 做出「n維歐氏幾何空間只允許有限多種兩兩不等價的空間群」；萊因哈特證明不規則多面體亦可填滿空間；托馬斯·黑爾斯於1998年提出了初步證明，並於2014年8月10日用計算機完成了克卜勒猜想的形式化證明，證明球體最緊密的排列是面心立方和六方最密兩種方式。
第19題	拉格朗日系統 (Lagrangian) 之解是否皆可解析	已解決	答案：是。1956年至1958年 <u>恩尼奧·德喬吉</u> 和 <u>約翰·福布斯·納什</u> 分別用不同方法證明。
第20題	所有邊值問題是否都有解	已解決	實際上工程和科研中遇到的邊值問題都是適定的，因而都可以確定是否有解。 [4]
第21題	證明有線性微分方程式有給定的單值群 (monodromy group)	已解決	此問題的答案取決於問題的表述：部分情況下是肯定的，部分情況下則是否定的。
第22題	將解析關係 (analytic relations) 以自守函數一致化	部分解決	1904年由 <u>保羅·克伯</u> 和 <u>龐加萊</u> 取得部分解決。詳見 <u>單值化定理</u> 。
第23題	變分法的長遠發展	開放性問題	包括希爾伯特本人、昂利·勒貝格、雅克·阿達馬等數學家皆投身於此。理察·貝爾曼提出的動態規劃可作為變分法的替代。

希爾伯特提出這些問題之後

- 開啟了一個數學的《大航海時代》
- 吸引來很多數學家，試圖解決這些問題

像是

- 羅素、哥德爾、Church、圖靈 . . .

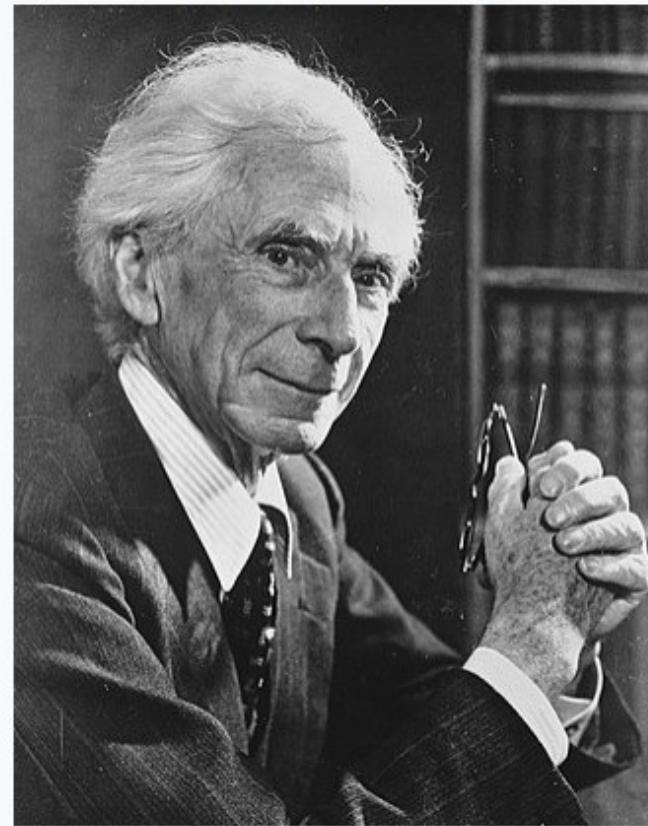
於是

- 羅素先登場了！

羅素

伯特蘭·羅素 

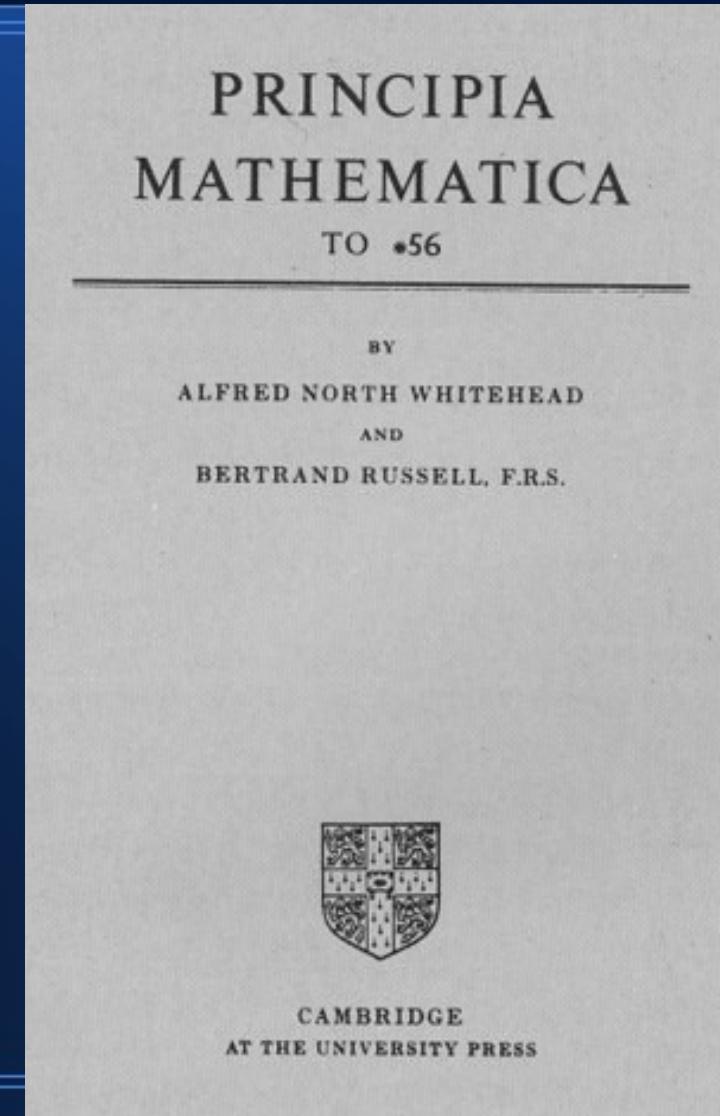
Bertrand Russell



1900 年

- 羅素正致力於撰寫一本
把數學系統公理化的書籍！

那本書稱為《數學原理》



數學原理

- 企圖將整個數學體系
建構成一棟穩固的大廈！

於是你会看到

- 羅素在試圖證明 $1+1=2$

*54·43. $\vdash \therefore \alpha, \beta \in 1. \supset : \alpha \cap \beta = \Lambda . \equiv . \alpha \cup \beta \in 2$

Dem.

$\vdash . *54\cdot26. \supset \vdash \therefore \alpha = \iota'x . \beta = \iota'y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y .$

$[*51\cdot231] \quad \equiv . \iota'x \cap \iota'y = \Lambda .$

$[*13\cdot12] \quad \equiv . \alpha \cap \beta = \Lambda \quad (1)$

$\vdash . (1) . *11\cdot11\cdot35. \supset$

$\vdash \therefore (\forall x, y) . \alpha = \iota'x . \beta = \iota'y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda \quad (2)$

$\vdash . (2) . *11\cdot54 . *52\cdot1. \supset \vdash . \text{Prop}$

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.

*54.43: "從這個命題將遵循,算術加法被定義時, $1 + 1 = 2$ 。 「卷I,第1版,379頁379
頁」 (頁面存檔備份,存於網際網路檔案館) (第二版362頁,362頁的刪節版)。(證明實際上是在卷II,完成第1版,86頁86頁,配以注釋完成的,「上面的命題是偶爾會有用的

問題是

- 到底要怎麼證明 $1+1=2$ 呢？

答案當然是

- 依靠一組公理系統
- 然後透過邏輯法則去推出來！

其中最重要的公理系統

- 就是《一階邏輯》了！

一階邏輯

• 得從符號開始學起

1. 量化符號 \forall 及 \exists

某些作者^[3]會把 \exists 符號定義為：

$$\exists x \mathcal{A} := \neg [\forall x (\neg \mathcal{A})],$$

如此只需要 \forall 做為基礎符號。

2. 邏輯聯結詞：以下為可能的表示符號，而波蘭表示法下的邏輯連接詞請參見邏輯運算的波蘭記法

- 否定： \neg 或 \sim
- 條件： \Rightarrow 或 \rightarrow 或 \supset
- 且： \wedge 或 $\&$
- 或： \vee 或 \parallel
- 雙條件： \Leftrightarrow 或 \leftrightarrow

某些作者^[4]會作如下的符號定義：

$$\mathcal{A} \wedge \mathcal{B} := \neg(\mathcal{A} \Rightarrow (\neg \mathcal{B})),$$

$$\mathcal{A} \vee \mathcal{B} := (\neg \mathcal{A}) \Rightarrow \mathcal{B},$$

$$\mathcal{A} \Leftrightarrow \mathcal{B} := (\mathcal{A} \Rightarrow \mathcal{B}) \wedge (\mathcal{B} \Rightarrow \mathcal{A}),$$

如此一來只需要否定和條件做為基礎符號。

然後用類似函數的語法

- 組成 predicate 與 function

斷言符號 [編輯]

「他們兩人是夫妻」，代表一個關於兩個"物件"的斷言，而「他是人」、「三點共線」顯示了斷言是容許一個—甚至多個物件，所以對於自然數 n 、 j 我們約定：

$$A_j^n(x_1, x_2, \dots, x_n)$$

為一階邏輯的合法詞彙，它在直觀上表示一個有 n 個"物件"的斷言，我們稱它為 n 元斷言符號。下標的自然數 j 只是拿來和其他同為 n 元的斷言符號作區別。

實用上只要有申明，不至於和其他詞彙引起混淆的話，我們可以用任意的形式簡寫一個斷言符號，如公理化集合論裡的 $x \in y$ 就是一個雙元斷言符號，也可以用 $A_1^2(x, y)$ 兀長地表達。

函數符號 [編輯]

「物體的顏色」、「夫妻的長子」說明了一列物件所唯一對應的物件。但不同的夫妻有不同的長子；不同的物體有不同的顏色，所以對於自然數 n 、 j 我們約定：

$$f_j^n(x_1, x_2, \dots, x_n)$$

為一階邏輯的合法詞彙，它在直觀上表示 n 個"物件"所對應到的東西，我們稱它為 n 元函數符號。但特別注意這種 "唯一對應" 的直觀想法，必須配上關於"等式"的性質(詳見下面等式定理)，才能被實現。

就像程式語言一樣

項 [編輯]

「那對夫妻的長子的職業」、「 $(x + y) \times z$ 」、「 $x \cup \emptyset$ 」代表變數可以與函數符號組成"更一般的物件"，為此我們遞迴地規定一類合法詞彙——項為：

1. 變數和常數是項。
2. 若 T_1, \dots, T_n 全都是項，那 $f_j^n(T_1, \dots, T_n)$ 也是項。
3. 項只能透過以上兩點，於有限步驟內建構出來。

我們習慣用大寫的西方字母(如英文字母、希伯來字母、希臘字母)代表項，如果變數不得不採用大寫字母，而跟會項引起混淆的話，需額外規定分辨的辦法。

只是改成邏輯的語言

原子公式 [\[編輯\]](#)

為了比較簡潔的規定甚麼是合式公式，我們先規定原子公式為：（若 T_1, \dots, T_n 是項）

$$A_j^n(T_1, \dots, T_n)$$

這樣的形式。

公式 [\[編輯\]](#)

一階邏輯的合式公式（簡稱公式或 *wf*）以下面的規則遞迴地定義：

1. 原子公式為公式。（美觀起見，在原子公式外面包一層括弧也是公式）
2. 若 \mathcal{A} 為公式，則 $(\neg \mathcal{A})$ 為公式。
3. 若 \mathcal{A} 與 \mathcal{B} 為公式，則 $(\mathcal{A} \Rightarrow \mathcal{B})$ 為公式。
4. 若 \mathcal{A} 為公式， x 為任意變數，則 $(\forall x \mathcal{A})$ 為公式。（美觀起見
 $(\forall x) \mathcal{A} := \forall x \mathcal{A}$ ，也就是裡面的量詞有無外包括弧都是公式）
5. 合式公式只能透過以上四點，於有限步驟內建構出來。

以及邏輯的代換法則

$$(\mathcal{A} \wedge \mathcal{B}) := \{\neg[\mathcal{A} \Rightarrow (\neg\mathcal{B})]\}$$

$$(\mathcal{A} \vee \mathcal{B}) := [(\neg\mathcal{A}) \Rightarrow \mathcal{B}]$$

$$(\mathcal{A} \Leftrightarrow \mathcal{B}) := [(\mathcal{A} \Rightarrow \mathcal{B}) \wedge (\mathcal{B} \Rightarrow \mathcal{A})]$$

$$(\exists x \mathcal{A}) := \{\neg[\forall x (\neg\mathcal{A})]\}$$
 (同樣美觀起見 $(\exists x) \mathcal{A} := \exists x \mathcal{A}$)

然後透過推論法則

在邏輯中，肯定前件（拉丁語：*Modus ponens*）是有效的、簡單的論證形式（常縮寫為MP）：
如果P，則Q；且P為真，故Q為真。

推理規則 [編輯]

MP律 [編輯]

對於公式 \mathcal{A} 和 \mathcal{B} ：

$\mathcal{A} \Rightarrow \mathcal{B}$ 和 \mathcal{A} 組合出 \mathcal{B} 。

直觀意義非常明顯，就是 $p \Rightarrow q$ 且 p 可以推出 q 。

肯定前件規則可以用相繼式符號寫為：

$$P \rightarrow Q, P \vdash Q$$

或用規則形式寫為：

$$\frac{P \rightarrow Q, P}{Q}$$

從三條邏輯公理

邏輯公理 [編輯]

如果 \mathcal{B} 、 \mathcal{C} 、 \mathcal{D} 都是公式則

- (A1) $\mathcal{B} \Rightarrow (\mathcal{C} \Rightarrow \mathcal{B})$
- (A2) $[\mathcal{B} \Rightarrow (\mathcal{C} \Rightarrow \mathcal{D})] \Rightarrow [(\mathcal{B} \Rightarrow \mathcal{C}) \Rightarrow (\mathcal{B} \Rightarrow \mathcal{D})]$
- (A3) $[(\neg \mathcal{B}) \Rightarrow (\neg \mathcal{C})] \Rightarrow [(\neg \mathcal{B} \Rightarrow \mathcal{C}) \Rightarrow \mathcal{B}]$

都是公理。

它們實際上是公理模式，代表著"跟自然數一樣多"條的公理。

與四條量詞公理

量詞公理 [\[編輯\]](#)

以下的 x 為任意變數， \mathcal{B} 、 \mathcal{C} 為任意公式。

- (A4) T 是一個項， t 為 T 中出現的任意變數；若 \mathcal{B} 裡，所有 $\forall t$ 的範圍裡都沒有自由的 x (這個情況稱為 \mathcal{B} 裡項 T 對 x 是自由的)，則

$$[(\forall x)\mathcal{B}(x)] \Rightarrow [\mathcal{B}(T)]$$

為公理

其中 $\mathcal{B}(T)$ 代表把 \mathcal{B} 裡自由的 x 都取代為 T 所得到的新公式。

- (A5) 如果 x 在 \mathcal{B} 裡完全被約束則

$$[(\forall x)(\mathcal{B} \Rightarrow \mathcal{C})] \Rightarrow [\mathcal{B} \Rightarrow (\forall x\mathcal{C})]$$

為公理

- (A6) $[(\forall x)(\mathcal{B} \Rightarrow \mathcal{C})] \Rightarrow [(\forall x\mathcal{B}) \Rightarrow (\forall x\mathcal{C})]$ 為公理

- (A7) 若 \mathcal{B} 是公理， x 是任意變數則

$$(\forall x)\mathcal{B}$$

也是公理。

就能進行演譯推論

- 像是證明 $\text{not}(\text{not } A) \Rightarrow A$

證明	[合併]
$\vdash \neg\neg A \Rightarrow A$	
(1) $(\neg A \Rightarrow \neg\neg A) \Rightarrow [(\neg A \Rightarrow \neg A) \Rightarrow A]$ (A3)	
(2) $\neg A \Rightarrow \neg A$ (I)	
(3) $(\neg A \Rightarrow \neg\neg A) \Rightarrow A$ (D2 with 1, 2)	
(4) $\neg\neg A \Rightarrow (\neg A \Rightarrow \neg\neg A)$ (A1)	
(5) $\neg\neg A \Rightarrow A$ (D1 with 3, 4)	

或者證明

- 若 $A \rightarrow B \Rightarrow \neg B \rightarrow \neg A$

(T2) Transposition-2

$$B \Rightarrow A \vdash \neg A \Rightarrow \neg B$$

證明	[合併]
(1) $\neg\neg B \Rightarrow B$ (DN)	
(2) $B \Rightarrow A$ (Hyp)	
(3) $\neg\neg B \Rightarrow A$ (D with 1, 2)	
(4) $A \Rightarrow \neg\neg A$ (DN)	
(5) $\neg\neg B \Rightarrow \neg\neg A$ (D1 with 3,4)	
(6) $(\neg\neg B \Rightarrow \neg\neg A) \Rightarrow (\neg A \Rightarrow \neg B)$ (T1, D)	
(7) $\neg A \Rightarrow \neg B$ (MP with 5, 6)	

我們可以從公理系統

- 透過嚴格的邏輯法則
推導出各種數學定理！

這些邏輯理論

- 並非《羅素》所發明的
- 而是承襲了《弗雷格》的一階邏輯
- 而《弗雷格》又承襲了《布林》的
布林邏輯體系

但是羅素到底要如何證明

- $1+1=2$ 呢？

這還缺了一點東西

- 那就是《皮亞諾公理系統》

皮亞諾公理系統

皮亞諾的這五條公理用非形式化的方法敘述如下：

1. 0是自然數；
2. 每一個確定的自然數 a ，都有一個確定的後繼數 a' ， a' 也是自然數；
3. 對於每個自然數 b 、 c ， $b=c$ 若且唯若 b 的後繼數= c 的後繼數；
4. 0不是任何自然數的後繼數；
5. 任意關於自然數的命題，如果證明：它對自然數0是真的，且假定它對自然數 a 為真時，可以證明對 a' 也真。那麼，命題對所有自然數都真。

其中，一個數的後繼數指緊接在這個數後面的數，例如，0的後繼數是1，1的後繼數是2等等；公理5保證了數學歸納法的正確性，從而被稱為歸納法原理。

皮亞諾公理的一階邏輯描述

- $\forall x(Sx \neq 0) \circ$
- $\forall x, y((Sx = Sy) \Rightarrow x = y) \circ$
- $(\varphi[0] \wedge \forall x(\varphi[x] \Rightarrow \varphi[Sx])) \Rightarrow \forall x(\varphi[x])$
- $\forall x(x + 0 = x) \circ$
- $\forall x, y(x + Sy = S(x + y)) \circ$
- $\forall x(x \cdot 0 = 0) \circ$
- $\forall x, y(x \cdot Sy = (x \cdot y) + x) \circ$

於是

- $\forall x(Sx \neq 0) \circ$
- $\forall x, y((Sx = Sy) \Rightarrow x = y) \circ$
- $(\varphi[0] \wedge \forall x(\varphi[x] \Rightarrow \varphi[Sx])) \Rightarrow \forall x(\varphi[x])$
- $\forall x(x + 0 = x) \circ$
- $\forall x, y(x + Sy = S(x + y)) \circ$
- $\forall x(x \cdot 0 = 0) \circ$
- $\forall x, y(x \cdot Sy = (x \cdot y) + x) \circ$

- 定義 : $1=S0$, $2=SS0$
- 然後你就可以證明
 - $S0+S0=S(S(0+0))=SS0$

所以羅素在數學原理一書中

- 終於證明了 $1+1=2 \dots$

但是當羅素開始想

- 《集合論》該怎麼用邏輯表達時

他發現了一個問題

- 用白話文說，就是：
 - 一個不包含自己的集合該怎麼定義？

羅素把它寫成數學

• 結果發現，死定了！

羅素悖論：設有一性質 P ，並以一性質函數表示： $P(x)$ ，且其中的自變量 x 有此特性： $x \notin x$ 。

現假設由性質 P 能夠確定一個滿足性質 P 的集合 A ——也就是說 $A = \{x \mid x \notin x\}$ 。那麼現在的問題是 $A \in A$ 是否成立？

首先，若 $A \in A$ ，則 A 是 A 的元素，那麼 A 具有性質 P ，由性質函數 P 可以得知 $A \notin A$ ；

其次，若 $A \notin A$ ，根據定義， A 是由所有滿足性質 P 的類組成，也就是說， A 具有性質 P ，所以 $A \in A$ 。

用白話文說

- $A = \{x \mid x \notin x\}$ 一個不包含自己的集合

$$A = \{x \mid x \notin x\}$$

- 於是若 A 包含自己， A 就不應該是 A 的成員，但若 A 不包含自己， A 就應該是 A 的成員

這個講法有點繞口

- 所以羅素發明了另一個悖論
- 稱為理髮師悖論！

理髮師悖論

- 有一個理髮師，他宣稱
 - 要為所有不自己理頭髮的人理頭髮
 - 但是不為任何自己理頭髮的人理頭髮
- 請問他做得到嗎？

這個問題

- 當然是做不到！

因為

- 他到底要不要為自己理頭髮呢？
 - 如果他自己理，那就違反第二條
 - 如果他不自己理，他就違反第一條

理髮師悖論

- 其實有個解套方法
- 假如那個理髮師是 Snoopy

那就沒有問題了！

當羅素發現這些悖論之後說

- 當我所建構的科學大廈即將完工之時，卻發現它的地基已經動搖了...

Hardly anything more unwelcome can befall a scientific writer than that one of the foundations of his edifice be shaken after the work is finished

於是羅素的數學原理

- 就很難寫下去了 ...

但是

- 這只是《集合論》上的一個小傷口
- 不代表《數論》上也會有同樣的問題或者《一階邏輯》本身也會有悖論！

終於

- 在 1929 年，傳來了好消息
- 哥德爾證明了一階邏輯不會有悖論，史稱《哥德爾完備定理》。

哥德爾完備定理

- 一階邏輯系統是一致且完備的，也就是所有的一階邏輯定理都可以透過機械性的推論程序證明出來，而且不會導出矛盾的結論。

Original proof of Gödel's completeness theorem

From Wikipedia, the free encyclopedia

The proof of Gödel's completeness theorem given by Kurt Gödel in his doctoral dissertation of 1929 (and a shorter version of the proof, published as an article in 1930, titled "The completeness of the axioms of the functional calculus of logic" (in German)) is not easy to read today; it uses concepts and formalisms that are no longer used and terminology that is often obscure. The version given below attempts to represent all the steps in the proof and all the important ideas faithfully, while restating the proof in the modern language of mathematical logic. This outline should not be considered a rigorous proof of the theorem.

Contents [hide]

- 1 Assumptions
- 2 Statement of the theorem and its proof
 - 2.1 Theorem 1. Every valid formula (true in all structures) is provable.
 - 2.2 Theorem 2. Every formula φ is either refutable or satisfiable in some structure.
 - 2.3 Equivalence of both theorems



Kurt Gödel (1925)

但是隔年

- 當哥德爾嘗試把數論的皮亞諾公設放進去時，一開始還算順利
- 不過當哥德爾把乘法符號也加進去時，卻發現了一個悖論！

為了陳述這個悖論

- 哥德爾寫下了長達 101 頁的論文
- 史稱《哥德爾不完備定理》

哥德爾不完備定理

在數理邏輯中，哥德爾不完備定理是庫爾特·哥德爾於1931年證明並發表的兩條定理。簡單地說，第一條定理指出：

任何自洽的形式系統，只要蘊涵皮亞諾算術公理，就可以在其中構造出體系中不能被證明的真命題，因此通過推理演繹不能得到所有真命題（即體系是不完備的）。

這是形式邏輯中的定理，容易被錯誤表述。有許多命題聽起來很像是哥德爾不完備定理，但事實上並不是。具體實例見對哥德爾定理的誤解。

把第一條定理的證明過程在體系內部形式化後，哥德爾證明了第二條定理。該定理指出：

任何邏輯自洽的形式系統，只要蘊涵皮亞諾算術公理，它就不能用於證明其本身的自洽性。

哥德爾不完備定理破壞了希爾伯特計劃的哲學企圖。大衛·希爾伯特提出，像實分析那樣較為複雜的體系的相容性，可以用較為簡單的體系中的手段來證明。最終，全部數學的相容性都可以歸結為基本算術的相容性。但哥德爾的第二條定理證明了基本算術的相容性不能在自身內部證明，因此當然就不能用來證明比它更強的系統的相容性了。

在這裡

- 我們當然無法寫下該定理的證明 …

但是

- 我們可以用程式人的想法寫下
哥德爾的思路邏輯！

哥德爾不完備定理的程式型

- 不存在一個程式，可以正確判斷一個
 - 「包含算術的一階邏輯字串」是否為定理...
- 要證明這件事，我們可以用反證法 !

首先假設

- 存在一個程式可以判斷某字串是否為定理

```
function Proveable(str)
    if (str is a theorem)
        return 1;
    else
        return 0;
end
```

然後提個問題

請問 $\text{isTheorem}(\exists s \text{-Provable}(s) \ \& \ \neg\text{Provable}(\neg s))$ 是否為真呢？

於是我們會發現有兩種可能

- 一種是無法證明所有定理（不完備）
- 另一種是會產生矛盾（不一致）

讓我們先用 T 代表 $\exists s \text{-Provabe}(s) \& \neg\text{Provabe}(\neg s)$ 這個邏輯式的字串，然後分別討論「真假」這兩個情況：

1. 如果 $\text{isTheorem}(T)$ 為真，那麼代表存在無法證明的定理，也就是 Provabe 函數沒辦法證明所有的定理。
2. 如果 $\text{isTheorem}(T)$ 為假，那麼代表 $\neg T$ 應該為真。這樣的話，請問 $\text{Provabe}(\neg T)$ 會傳回甚麼呢？讓我們分析看看：

```
function Provabe(-T)
    if (-T is a theorem) // 2.1 這代表  $\neg(\exists s \text{-Provabe}(s) \& \neg\text{Provabe}(\neg s))$  是個定理，也就是  $\text{Provabe}()$  可以正確證明所有定理。
        return 1;           // 但這樣的話，就違反了上述「2. 如果  $\text{isTheorem}(T)$  為假」的條件了。
    else
        return 0;           // 2.2 否則代表  $\neg T$  不是個定理，也就是存在  $(\exists)$  某些定理  $s$  是無法證明的。
    end
    // 但這樣的話，又違反上述「2. 如果  $\text{isTheorem}(T)$  為假」的條件了。
```

於是我們斷定：如果 $\text{Provabe}()$ 對所有輸入都判斷正確的話，那麼 2 便是不可能的，因為 (2.1, 2.2) 這兩條路都違反 2 的假設，也就是只有 1 是可能的，所以我們可以斷定 $\text{Provabe}(s)$ 沒辦法正確證明所有定理。

所以我們得到了結論

- 不存在一個程式，可以正確判斷一個
 - 「包含算術的一階邏輯字串」
- 是否為定理！

這樣的結論

- 是在還沒有電腦，沒有程式的年代
透過純粹的數學推論證明出來的！
- 這是為何必須寫上 101 頁的原因

第二個故事

程式

當哥德爾證明了以下兩個定理之後

- 哥德爾完備定理
- 哥德爾不完備定理

數學家

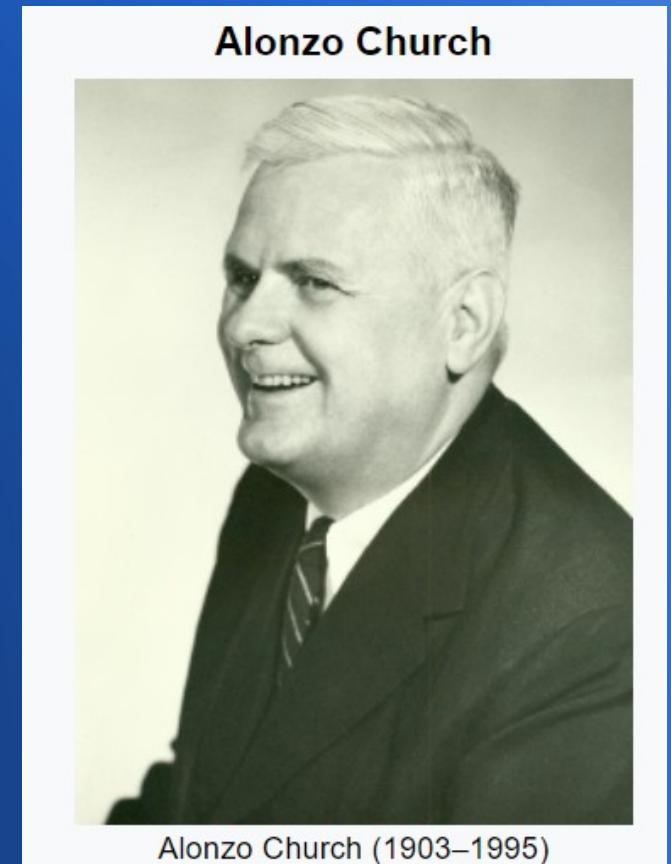
- 還是有事可以做！

因為

- 那個年代還沒有電腦可以玩
也沒電視可以看
不想數學要幹嘛呢？

於是有一個人

- 他叫做 Alonzo Church
- 發明了一種稱為 λ Calculus 的數學！



λ Calculus 長這樣

- 語法

Syntax	Name	Description
x	Variable	A character or string representing a parameter or mathematical/logical value.
$(\lambda x.M)$	Abstraction	Function definition (M is a lambda term). The variable x becomes bound in the expression.
$(M N)$	Application	Applying a function to an argument. M and N are lambda terms.

- 運算

Operation	Name	Description
$(\lambda x.M[x]) \rightarrow (\lambda y.M[y])$	α -conversion	Renaming the bound variables in the expression. Used to avoid name collisions .
$((\lambda x.M) E) \rightarrow (M[x := E])$	β -reduction	Replacing the bound variables with the argument expression in the body of the abstraction.

放大一點

語法

Syntax	Name
x	Variable
$(\lambda x.M)$	Abstraction
$(M N)$	Application

運算

Operation
$(\lambda x.M[x]) \rightarrow (\lambda y.M[y])$
$((\lambda x.M) E) \rightarrow (M[x := E])$

看來很抽象

- 其實就是匿名函數的定義與呼叫

語法

Syntax	Name
x	Variable
$(\lambda x.M)$	Abstraction
$(M N)$	Application

運算

Operation
$(\lambda x.M[x]) \rightarrow (\lambda y.M[y])$
$((\lambda x.M) E) \rightarrow (M[x := E])$

如果用現代的語法

- 就像這樣

$$\begin{aligned}((x, y) \mapsto x^2 + y^2)(5, 2) \\= 5^2 + 2^2 \\= 29,\end{aligned}$$

也可以只傳部分參數進去

- 這就是函數呼叫的 curry 技巧

$$\left((x \mapsto (y \mapsto x^2 + y^2))(5) \right)(2)$$

$= (y \mapsto 5^2 + y^2)(2)$ // the definition of x has been used with 5 in the inner expression. This is like β -reduction.

$= 5^2 + 2^2$ // the definition of y has been used with 2. Again, similar to β -reduction.

$= 29$

但是

- Church 厲害的地方是
將一切都用 λ calculus 定義
- 包含 0, 1, 2, true, false, if,
等等程式語言的元素

以下這篇文章

- 只要你靜下心來看，會發現 λ -Calculus 說明得很清楚

The screenshot shows a GitHub repository page for 'sgillespie/lambda-calculus'. The page title is 'lambda-calculus'. Below the title, there's a large heading 'Lambda Calculus'. A paragraph explains that Lambda Calculus is a tiny functional language for expressing computation based on function abstraction and application. It notes that its ideas form the basis of nearly all functional programming languages like ML, Haskell, and Scheme. The text also mentions that Lambda Calculus is a very small language and a good starting point for studying functional programming language design and implementation.

Below this, another paragraph discusses the features of Lambda Calculus, mentioning that it has few peculiar features: all objects are functions, every function accepts functions as arguments and returns functions, pure λ is untyped, and λ 's functions can only take one argument.

Following this, an example is given: consider a function that adds three numbers. The code shown is:

```
add x y z = x + y + z
```

In the text, it says that in λ , we would need to compose several functions to simulate a function that takes more than one argument. The code shown for this is:

```
 $\lambda x. \lambda y. \lambda z. x + y + z$ 
```

<https://github.com/sgillespie/lambda-calculus/blob/master/doc/lambda-calculus.md>

讓我們從頭開始

- 認識 λ Calculus 的語法和語義

λ Calculus 的語法

- 變數 Variable
 - $x, y, z, a, b, c \dots$
- 函數定義 λ abstraction
 - $\lambda v. \text{body}$
- 函數呼叫 Function Application
 - $f x y z$ (其中 f 是函數)

變數綁定

- $\lambda \ X. \ X$
 - X 變數被綁定了
- $\lambda \ y. \ f \ y$
 - y 被綁定， f 沒有

函數範例

- $\lambda x. x$
 - 單位函數 $f(x)=x$
- $\lambda x. \lambda y. x$
 - 傳回第一個參數 $f(x, y)=x$
- $\lambda x. \lambda y. y$
 - 傳回第二個參數 $f(x, y)=y$

Curry

- λ 後面只能跟著一個變數，不允許多個參數，要模擬多參數函數，就必須要能傳回少一個參數的函數
- 這個手法稱為 currying

多參數函數

- $\lambda f. \lambda x. \lambda y. f\ x\ y$

可以縮寫成

$$\lambda f\ x\ y. f\ x\ y$$

於是 Church Numerals 誕生了

- 用函數代表數值 0, 1, 2, 3...
- 0: $\lambda f x. x$
- 1: $\lambda f x. f x$
- 2: $\lambda f x. f(f x)$
- 3: $\lambda f x. f(f(f x))$

這時你的心裡一定會想：幹，這三小！

然後我們可以定義 succ 函數

```
zero: λ f x. x  
succ: λ n f x. f (n f x)
```

We have defined natural numbers inductively, using two cases. Zero is defined exactly as we did earlier, and its successor, as $n + 1$. Every natural can be composed using those two functions. For example, we can achieve the number 2, by using only zero and successor.

```
2 = succ (succ zero)  
= (λ n f x. f (n f x)) ((λ n f x. f (n f x)) λ f x. x)  
= λ f x. f (f x)
```

有了以上的語法

- 我們就可以定義《運算》了

λ Calculus 有三種運算

- Beta reduction
- Alpha conversion
- Eta conversion

Beta Reduction

- 函數套用 : $(\lambda \ x. \ \text{body}) \ y$ 將 body 裏的 x 用 y 取代
- 範例
 - 0: $(\lambda \ f \ x. \ x)$
 - succ: $(\lambda \ n \ f \ x. \ f \ (n \ f \ x))$
 - $\text{succ}(0) = \text{succ} \ (\lambda \ f \ x. \ x)$
 $\rightarrow \lambda \ f \ x. \ f \ ((\lambda \ f \ x. \ x) \ f \ x)$
 $\rightarrow \lambda \ f \ x. \ f \ x$
 - $\rightarrow 1$

Alpha Conversion

- 當變數名稱相衝時，需要改名

Name captures occur when the free variables of an argument have a parameter of the same name in the body of the abstraction. Using α -conversion, we change the names of parameter bound by the abstraction [6]. This means that the following expressions are equal.

```
(λ x. x)  
(λ y. y)
```

Returning to our capture example, we first look for captures.

```
(λ f x. f x) (λ f. x)
```

Because `x` is free in the argument `(λ f. x)`, we need to use α -conversion to change the parameter `x` to a unique name.

```
(λ f x. f x) (λ f. x) → (λ f y. f y) (λ f. x)
```

We now proceed with B-reduction.

```
(λ f y. f y) (λ f. x) → λ y. (λ f. x) y  
→ λ y. x
```

Eta Conversion

- 當某參數綁定卻沒影響時，可簡化

η -conversion allows us to further eliminate abstractions if they give the same result for any argument [1]. η -conversion allows us to reduce the following expression

$$\lambda x. f x \rightarrow f$$


as long as x does not occur free in f .

有了以上這些

- 我們就可以寫 λ Calculus 程式了

怎麼寫？

像是這樣

```
0: λ f x. x  
1: λ f x. f x  
2: λ f x. f (f x)  
3: λ f x. f (f (f x))
```

Again, these can be all be defined inductively

```
zero: λ f x. x  
succ: λ n f x. f (n f x)
```

We now define addition and subtraction [1]:

```
add: λ m n f x. m f (n f x)  
multiply: λ m n f. m (n f)
```

然後你就可以運算

```
add: λ m n f x. m f (n f x)  
multiply: λ m n f. m (n f)
```



For example, consider the expression $2 + 3$. We will solve this using the three λ reduction rules. We begin by expanding $\text{add } 2 \ 3$ with the definitions above

```
add 2 3 → (λ m n f x. m f (n f x)) (λ f x. f (f x)) (λ f x. f (f (f x)))
```

We now apply B-reduction twice, beginning with the left, outermost application

```
→ (λ n f x. (λ f x. f (f x)) f (n f x)) (λ f x. f (f (f x)))  
→ λ f x. (λ f x. f (f x)) f ((λ f x. f (f (f x))) f x)
```

運算 . . .

```
→ λ f x. (λ g x. g (g x)) f ((λ f x. f (f (f x))) f x)
```

Now that the capture has been resolved, we continue with B-reduction.

```
→ λ f x. (λ x. f (f x)) ((λ f x. f (f (f x))) f x)
```

Once again, we apply from the left, outermost abstraction, applying $\lambda x. f (f x)$ to the massive argument $\lambda f x. f (f (f x))) f x$. However, there's another capture, x . We α -convert that, then B-reduce it again.

```
→ λ f x. (λ y. f (f y)) ((λ f x. f (f (f x))) f x)  
→ λ f x. (f (f ((λ f x. f (f (f x))) f x)) f x)
```

再運算 . . .

We repeat this process until there are no more redexes

```
→ λ f x. (f (f ((λ g y. g (g (g y)))) f x)  
→ λ f x. (f (f (f (f (f x)))))
```

From the sequence above, we know this is 5. Since $2 + 3 = 5$, we have arrived at the correct result.

終於得到

0: $\lambda f x. x$
1: $\lambda f x. f x$
2: $\lambda f x. f (f x)$
3: $\lambda f x. f (f (f x))$

add 2 3 $\rightarrow (\lambda m n f x. m f (n f x)) (\lambda f x. f (f x)) (\lambda f x. f (f (f x)))$

Again, these can be all be defined inductively

zero: $\lambda f x. x$
succ: $\lambda n f x. f (n f x)$

We repeat this process until there are no more redexes

$\rightarrow \lambda f x. (f (f ((\lambda g y. g (g (g y)))) f x))$
 $\rightarrow \lambda f x. (f (f (f (f (f x)))))$

We now define addition and subtraction [1]:

add 2 3 = 2+3 = 5

add: $\lambda m n f x. m f (n f x)$
multiply: $\lambda m n f. m (n f)$

其實

- 你可以用 JavaScript
做到 λ -Calculus 所做的事

Guilherme J. Tramontina 就這樣做了

← → C github.com/gtramontina/lambda/blob/master/lambda.js

211 lines (159 sloc) | 5.96 KB

```
1 // Arithmetics -----
2
3 IDENTITY      = x => x
4 SUCCESSOR      = n => f => x => f(n(f))(x)
5 PREDECESSOR    = n => f => x => n(g => h => h(g(f)))(_ => x)(u => u)
6 ADDITION       = m => n => n(SUCCESSOR)(m)
7 SUBTRACTION    = m => n => n(PREDECESSOR)(m)
8 MULTIPLICATION = m => n => f => m(n(f))
9 POWER          = x => y => y(x)
10 ABS_DIFFERENCE = x => y => ADDITION(SUBTRACTION(x)(y))(SUBTRACTION(y)(x))
11
12 // Logic -----
13
14 TRUE   = t => f => t
15 FALSE  = t => f => f
16 AND    = p => q => p(q)(p)
17 OR     = p => q => p(p)(q)
18 XOR    = p => q => p(NOT(q))(q)
19 NOT    = c => c(FALSE)(TRUE)
20 IF     = c => t => f => c(t)(f)
21
```

你可以用 node.js 執行

```
> node lambda.js
[✓] TRUE
[✓] FALSE
[✓] AND
[✓] OR
[✓] XOR
[✓] NOT
[✓] IF
[✓] IDENTITY
[✓] SUCCESSOR
[✓] PREDECESSOR
[✓] ADDITION
[✓] SUBTRACTION
[✓] MULTIPLICATION
[✓] POWER
[✓] ABS_DIFFERENCE
[✓] IS_ZERO
[✓] IS_LESS_THAN
[✓] IS_LESS_THAN_EQUAL
[✓] IS_EQUAL
[✓] IS_NOT_EQUAL
[✓] IS_GREATER_THAN_EQUAL
[✓] IS_GREATER_THAN
[✓] IS_NULL

--- Examples ---

[✓] FACTORIAL: 5! = 120
[✓] FIBONACCI: 10 = 55
```

不過

- Church 是數學家，當然不會只做到這樣

Church 在 1936 年證明了

- λ -Calculus 裏有些
《不可解的問題》
(Unsolvable Problem)

Church 的論文



An Unsolvable Problem of Elementary Number Theory

Alonzo Church

American Journal of Mathematics, Vol. 58, No. 2. (Apr., 1936), pp. 345-363.

Stable URL:

<http://links.jstor.org/sici?&sici=0002-9327%28193604%2958%3A2%3C345%3AAUPOEN%3E2.0.CO%3B2-1>

American Journal of Mathematics is currently published by The Johns Hopkins University Press.

這是論文的主定理

- 要寫一個函數判斷某運算式有沒有正規形式是不可能的

THEOREM XVIII. *There is no recursive function of a formula \mathbf{C} , whose value is 2 or 1 according as \mathbf{C} has a normal form or not.*

That is, the property of a well-formed formula, that it has a normal form, is not recursive.

For assume the contrary.

Then there exists a recursive function H of one positive integer such that $H(m) = 2$ if m is the Gödel representation of a formula which has a normal form, and $H(m) = 1$ in any other case. And, by Theorem XVI, H is λ -definable by a formula \mathfrak{H} .

第三個故事

機器

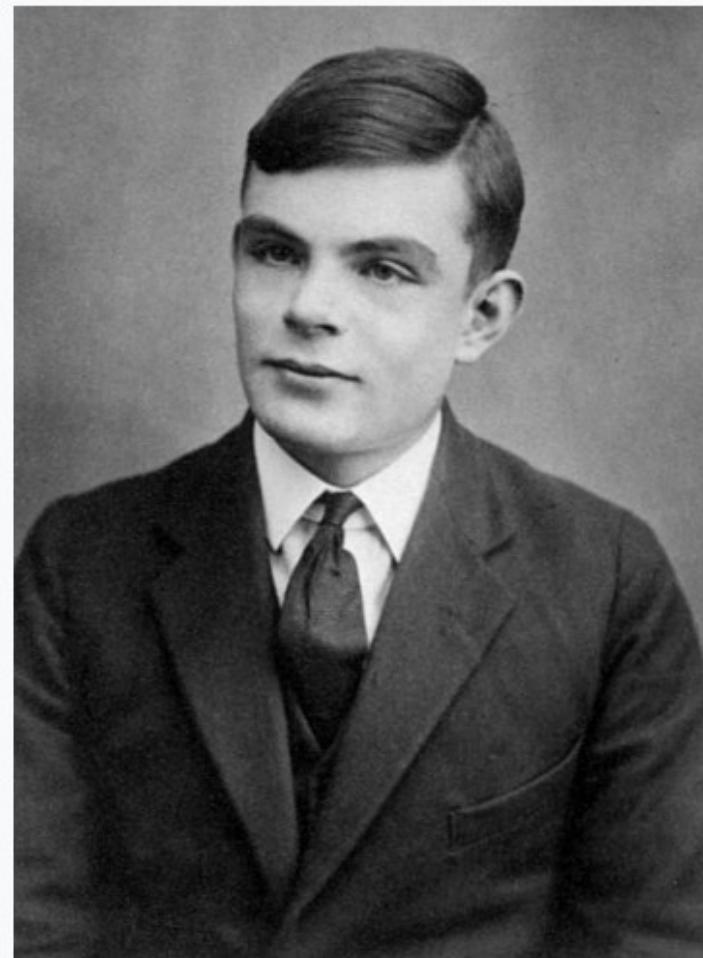
就在 Church 證明發表的那年

- 也就是 1936 年，圖靈 (Alan Turing) 證明了一個定理
- 那就是《停機問題》(Halting Problem) 是不可解的！

圖靈 (Alan Turing)

Alan Turing

OBE FRS PHD



Turing c. 1928 at age 16

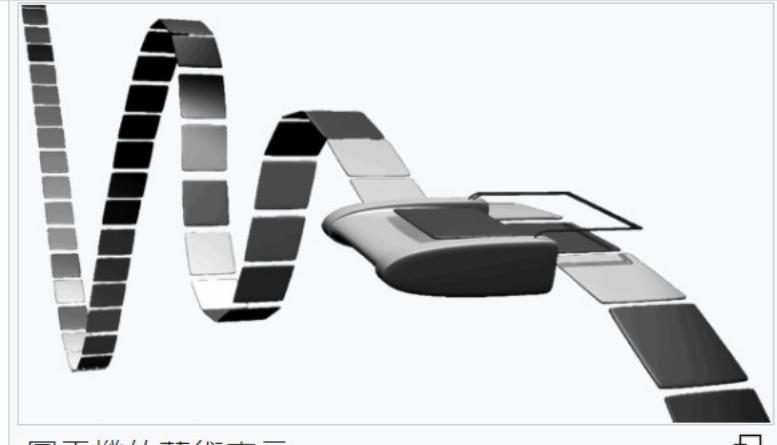
不過圖靈的證明

- 是建構在圖靈機之上的

圖靈機（英語：Turing machine），又稱確定型圖靈機，是英國數學家艾倫·圖靈於1936年提出的一種將人的計算行為抽象化的數學邏輯機，其更抽象的意義為一種計算模型，可以看作等價於任何有限邏輯數學過程的終極強大邏輯機器。

目次 [隱藏]

- 1 圖靈的基本思想
- 2 圖靈機的正式定義
- 3 圖靈機的基本術語
- 4 圖靈機的例子
- 5 通用圖靈機



圖靈機的藝術表示

圖靈機

- 是一台可讀寫磁帶的機器

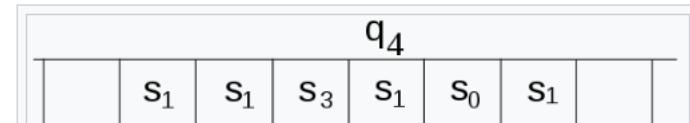
圖靈的基本思想是用機器來類比人們用紙筆進行數學運算的過程，他把這樣的過程看作下列兩種簡單的動作：

- 在紙上寫上或擦除某個符號；
- 把注意力從紙的一處移動到另一處；

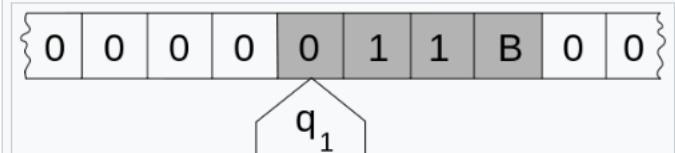
而在每個階段，人要決定下一步的動作，依賴於 (a) 此人當前所關注的紙上某個位置的符號和 (b) 此人當前思維的狀態。

為了類比人的這種運算過程，圖靈構造出一台假想的機器，該機器由以下幾個部分組成：

1. 一條無限長的紙帶**TAPE**。紙帶被劃分為一個接一個的小格子，每個格子上包含一個來自有限字母表的符號，字母表中有一個特殊的符號□表示空白。紙帶上的格子從左到右依次被編號為0, 1, 2, …，紙帶的右端可以無限伸展。
2. 一個讀寫頭**HEAD**。它可以在紙帶上左右移動，能讀出當前所指的格子上的符號，並能改變它。
3. 一套控制規則**TABLE**。它根據當前機器所處的狀態以及當前讀寫頭所指的格子上的符號來確定讀寫頭下一步的動作，並改變狀態暫存器的值，令機器進入一個新的狀態，按照以下順序告知圖靈機命令：
 - 1. 寫入（替換）或擦除當前符號；
 - 2. 移動 **HEAD**，'L'向左，'R'向右或者'N'不移動；
 - 3. 保持當前狀態或者轉到另一狀態。
4. 一個狀態暫存器。它用來儲存圖靈機當前所處的狀態。圖靈機的所有可能狀態的數目是有限的，並且有一個特殊的狀態，稱為停機狀態。參見[停機問題](#)。



在某些模型中，紙帶移動，而未用到的紙帶真正是「空白」的。要進行的指令 (q4) 展示在掃描到方格之上 (由Kleene (1952) p.375繪製)。



在某些模型中，讀寫頭沿著固定的紙帶移動。要進行的指令 (q1) 展示在讀寫頭內。在這種模型中「空白」的紙帶是全部為0的。有陰影的方格，包括讀寫頭掃描到的空白，標記了1,1,B的那些方格，和讀寫頭符號，構成了系統狀態。（由Minsky (1967) p.121繪製）

但圖靈用數學描述它

一台圖靈機是一個七元有序組 $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ ，其中 Q, Σ, Γ 都是有限集合，且滿足：

1. Q 是非空有窮狀態集合；
2. Σ 是非空有窮輸入字母表，其中不包含特殊的空白符□；
3. Γ 是非空有窮帶字母表且 $\Sigma \subset \Gamma$ ；
4. $\square \in \Gamma - \Sigma$ 為空白符，也是唯一允許出現無限次的字元；
5. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, -\}$ 是轉移函數，其中 L, R 表示讀寫頭是向左移還是向右移，- 表示不移動；
6. $q_0 \in Q$ 是起始狀態；
7. $q_{accept} \in Q$ 是接受狀態。 $q_{reject} \in Q$ 是拒絕狀態，且 $q_{reject} \neq q_{accept}$ 。

圖靈機 $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ 將以如下方式運作：

開始的時候將輸入符號串 $\omega = \omega_0 \omega_1 \dots \omega_{n-1} \in \Sigma^*$ 從左到右依此填在紙帶的第 $0, 1, \dots, n-1$ 號格子上，其他格子保持空白（即填以空白符□）。 M 的讀寫頭指向第 0 號格子， M 處於狀態 q_0 。機器開始執行後，按照轉移函數 δ 所描述的規則進行計算。例如，若當前機器的狀態為 q ，讀寫頭所指的格子中的符號為 x ，設 $\delta(q, x) = (q', x', L)$ ，則機器進入新狀態 q' ，將讀寫頭所指的格子中的符號改為 x' ，然後將讀寫頭向左移動一個格子。若在某一時刻，讀寫頭所指的是第 0 號格子，但根據轉移函數它下一步將繼續向左移，這時它停在原地不動。換句話說，讀寫頭始終不移出紙帶的左邊界。若在某個時刻 M 根據轉移函數進入了狀態 q_{accept} ，則它立刻停機並接受輸入的字串；若在某個時刻 M 根據轉移函數進入了狀態 q_{reject} ，則它立刻停機並拒絕輸入的字串。

注意，轉移函數 δ 是一個部分函數，換句話說對於某些 q, x ， $\delta(q, x)$ 可能沒有定義，如果在執行中遇到下一個操作沒有定義的情況，機器將立刻停機。

因為當時

- 根本就沒有電腦
- 圖靈只能用數學描述他的機器

然後用數學證明

- 停機問題是不可解的

停機問題（英語：halting problem）是邏輯數學中可計算性理論的一個問題。通俗地說，停機問題就是判斷任意一個程式是否能在有限的時間之內結束執行的問題。該問題等價於如下的判定問題：是否存在一個程式P，對於任意輸入的程式w，能夠判斷w會在有限時間內結束或者無窮迴圈。

艾倫·圖靈在1936年用對角論證法證明了，不存在解決停機問題的通用演算法。這個證明的關鍵在於對電腦和程式的數學定義，這被稱為圖靈機。停機問題在圖靈機上是不可判定問題。這是最早提出的決定性問題之一。

用數學語言描述，則其本質問題為：給定一個圖靈機T，和一個任意語言集合S，是否T會最終停機於每一個 $s \in S$ 。其意義相同於可確定語言。顯然任意有限 S 是可判定性的，可數的（countable）S 也是可停機的。

停機問題包含了自我指涉，本質是一階邏輯的不完備性，類似的命題有理髮師悖論、全能悖論等。

這是圖靈的論文

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO
THE ENTScheidungsproblem

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The “computable” numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

後來二次大戰爆發

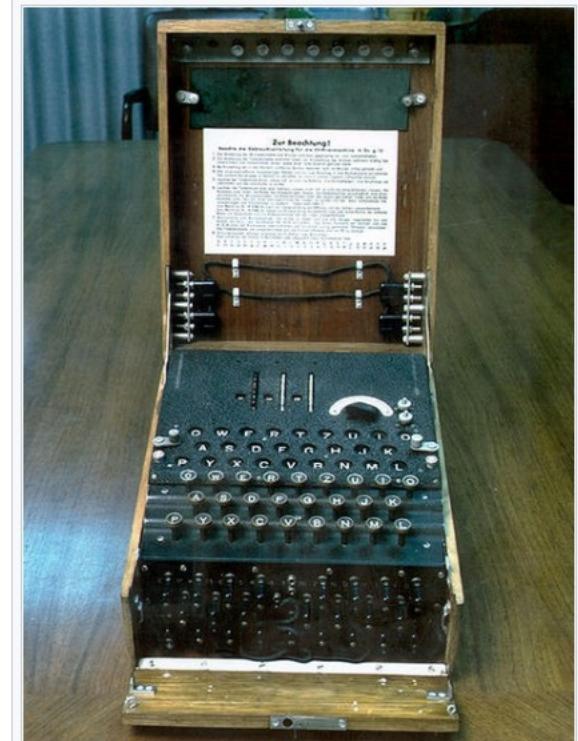
- 圖靈參與了《破解德軍密碼》的計畫

德軍所使用的 Enigma 密碼機

恩尼格瑪密碼機（德語：Enigma，又譯恩尼格密碼機、啞謎機、奇謎機^[1]或謎式密碼機）是一種用於加密與解密檔案的密碼機。確切地說，恩尼格瑪是對二戰時期納粹德國使用的一系列相似的旋轉機加解密機器的統稱，它包括了許多不同的型號，為密碼學對稱加密演算法的流加密。

20世紀20年代早期，恩尼格瑪密碼機開始應用於商業，一些國家的軍隊與政府也使用過該密碼機，密碼機的主要使用者包括第二次世界大戰時的納粹德國。^[2]

在恩尼格瑪密碼機的所有版本中，最著名的是德國使用的軍用版本。儘管此機器的安全性較高，但盟軍的密碼學家們還是成功地破解了大量由這種機器加密的訊息。1932年，波蘭密碼學家馬里安·雷耶夫斯基、傑爾茲·羅佐基和亨里克·佐加爾斯基根據恩尼格瑪機的原理破解了它。1939年中期，波蘭政府將此破解方法告知了英國和法國，但直到1941年英國海軍擷取德國U-110潛艇，得到密碼機和密碼本後才成功破解。密碼的破解使得納粹海軍對英美商船補給船的大量攻擊失效。盟軍的情報部門將破解出來的密碼稱為ULTRA，ULTRA極大地幫助了西歐的盟軍部隊。關於ULTRA到底對戰爭有多大貢獻尚存爭論，但普遍認為盟軍在西歐的勝利能夠提前兩年，完全是因為恩尼格瑪密碼機被成功破解的緣故。^{[3][4]}



一台德國軍用三旋轉盤恩尼格瑪
密碼機的接線板、鍵盤、顯示板和旋
轉盤。

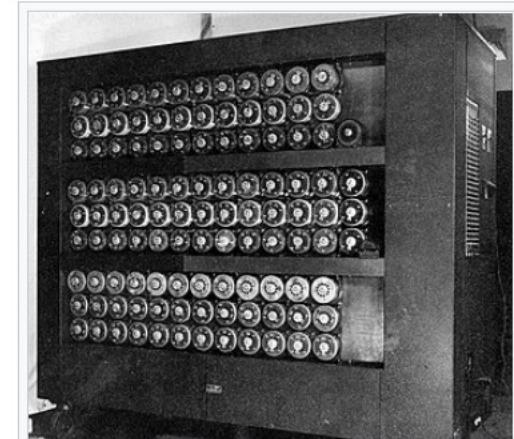
被圖靈的 Bombe 機破解

This article is about the decryption device used at Bletchley Park. For the earlier Polish decryption device, see [Bomba \(cryptography\)](#). For the European dessert of the same name, see [Bombe glacée](#). For other uses, see [Bombe \(disambiguation\)](#).

The **bombe** (UK: /bɒmb/) was an [electro-mechanical](#) device used by British [cryptologists](#) to help decipher German [Enigma-machine](#)-encrypted secret messages during [World War II](#).^[1] The [US Navy](#)^[2] and [US Army](#)^[3] later produced their own machines to the same functional specification, albeit engineered differently both from each other and from Polish and British bombes.

The British bombe was developed from a device known as the "bomba" (Polish: *bomba kryptologiczna*), which had been designed in Poland at the [Biuro Szyfrów](#) (Cipher Bureau) by cryptologist [Marian Rejewski](#), who had been breaking German [Enigma](#) messages for the previous seven years, using it and earlier machines. The initial design of the British bombe was produced in 1939 at the UK [Government Code and Cypher School](#) (GC&CS) at [Bletchley Park](#) by [Alan Turing](#),^[4] with an important refinement devised in 1940 by [Gordon Welchman](#).^[5] The engineering design and construction was the work of [Harold Keen](#) of the [British Tabulating Machine Company](#). The first bombe, code-named *Victory*, was installed in March 1940^[6] while the second version, *Agnes Dei* or *Agnes*, incorporating Welchman's new design, was working by August 1940.^[7]

The bombe was designed to discover some of the daily settings of the Enigma machines on the various German military [networks](#): specifically, the set of rotors in use and their positions in the machine; the rotor core start positions for the message—the message [key](#)—and one of the wirings of the plugboard.^{[8][9][10]}

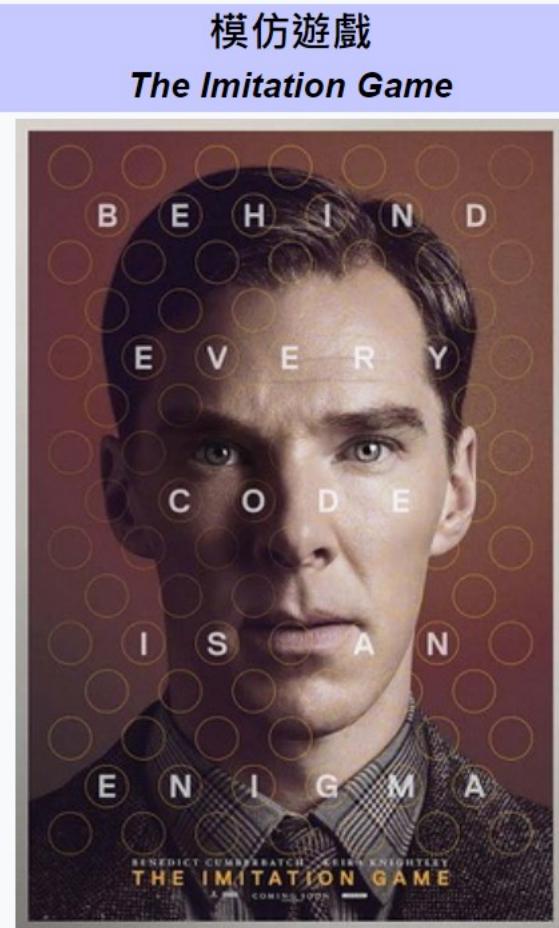


A wartime picture of a [Bletchley Park Bombe](#)

這個故事後來被拍成電影

《模仿遊戲》（英語：*The Imitation Game*，香港譯《解碼遊戲》）是一部2014年英美合拍的歷史劇情片。講述英國數學家、邏輯學家、密碼分析學家和計算機科學家艾倫·圖靈在二戰中幫助盟軍破譯納粹德國的軍事密碼的真實故事[12][13][14]。挪威導演摩頓·帝敦執導，編劇格雷厄姆·摩爾改編自安德魯·霍奇斯所寫的傳記《Alan Turing: The Enigma》[15][16][17][18]，英國演員班奈狄克·康柏拜區任男主角，其他主演包括綺拉·奈特莉、馬修·古迪和馬克·史壯。

二戰期間，英國劍橋大學的教授艾倫·圖靈（班奈狄克·康柏拜區飾）獲軍情六處秘密任命，與一群專家組成解密組，試圖破解由納粹德國獨創，號稱世上最精密的情報機器——「Enigma」密碼機。圖靈獨具才能，被邱吉爾任命為組內的領導人，但其古怪的脾氣卻使他受到組員的抵制，過程遭遇重重挫折，幸好在聰明靚麗的新成員瓊恩·克拉克（綺拉·奈特莉飾）的鼓勵下，圖靈主動打破和同僚的隔閡，最終合力研發出破譯地方機密的裝置，超過1400萬人得以避開戰火，而世上電腦的雛形亦宣告在那刻誕生。然而戰後多年，圖靈卻被揭發自己的同性戀傾向，而被英國政府入罪...



電影海報

二次大戰末期

• 電腦開始被製造出來

機械和電子計算機器從19世紀就開始出現了，但是20世紀30、40年代被看作是現代計算機時代的開端。

- 德國Z3計算機（1941年5月公布）是康拉德·楚澤設計的。這是第一台通用的數字計算機。但是它是機電計算機，而不是電子計算機，因為所有功能都使用繼電器。它使用二進位數學進行邏輯地計算。它可用打孔紙帶編程，但是沒有邏輯分支。儘管當初設計的不是圖靈完全的，但是在1998年人們發現它是圖靈完全的（但是如果要利用這種圖靈完全性質，需要複雜、聰明的破解）。1943年，這台計算機在柏林毀於轟炸襲擊。
- 美國阿塔納索夫-貝瑞計算機（ABC，1941年夏天公布）是第一台電子計算設備。它使用真空管實現了二進位計算，但是不是通用的，而是僅僅用於求解線性方程組。這台計算機也沒有利用電子計算的速度優勢。有兩方面限制了它的速度：一是旋轉電容鼓存儲器，另一個是它的輸入輸出系統要把中間結果寫出到紙片上。這台計算機是手動控制的，並且不可程式。
- 英國的巨人計算機（Colossus computer，1943年用於密碼分析）是湯米·弗勞爾設計的。這些計算機是數字的、電子的，可用插板和開關編程，但是僅用於密碼破譯，並不是通用的^[25]。
- 霍華德·艾肯在1944年設計的馬克一號電腦使用繼電器，可用打孔紙帶編程。可以計算一般的數學函數，但是沒有分支結構。
- 就像Z3和馬克一號一樣，ENIAC可以運行任意數學運算序列，但是並不是從紙帶上讀取數據。像Colossus一樣，它可以用插板和開關編程。ENIAC將全面、圖靈完全的可程式能力與電子計算的高速性結合在一起。

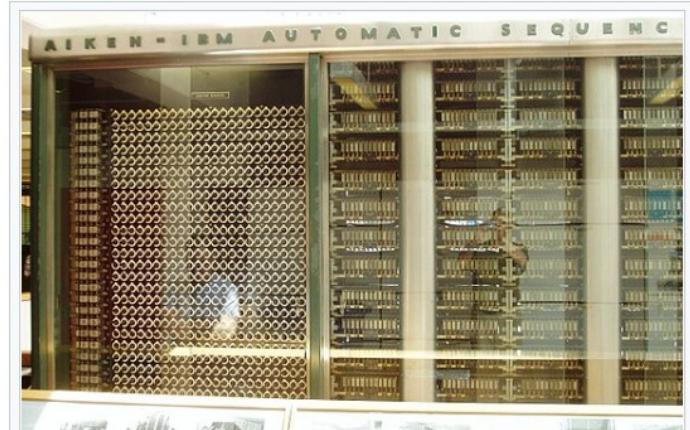
像是 1944 年的 MARK I

馬克一號（Mark I）是美國第一部大尺度自動數位電腦，被認為是第一部萬用型計算機。它的生產和設計者給它起的名字是全自動化循序控制計算機（Automatic Sequence Controlled Calculator，縮寫為ASCC），馬克一號是它的使用者哈佛大學給它起的名字。

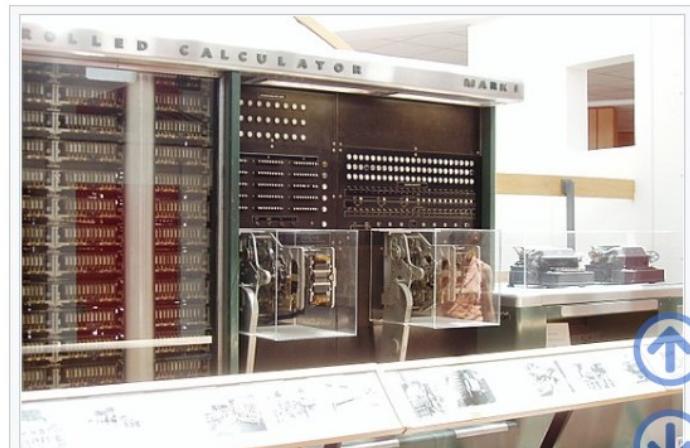
這部機電式ASCC是由IBM的霍華德·艾肯所設計的，在1944年8月7日搬到哈佛大學。馬克一號的特點為全自動運算。一旦開始運算便無須人為介入。馬克一號是第一部被實作出來的全自動電腦，同時與當年的其他電子式電腦相比它非常可靠。大家認為「這是現代電腦時代的開端」以及「真正的電腦時代的曙光」。

ASCC是由開關、繼電器、轉軸以及離合器所構成。它使用了765,000個元件以及幾百哩長的電線，組裝大小為16公尺（51呎）長，2.4公尺（8呎）高，2呎深。重達4500公斤（5短噸）。其基本計算單元使用同步式機械，所以它有一根長15公尺（50呎）的傳動軸，並由一顆4千瓦的馬達所驅動。馬可一號可以儲存72組數據，每組數據有23位十進位數字。每秒可執行3次加法或是減法。一個乘法則須6秒，一個除法須15.3秒，計算一個對數或是一個三角函式需花費超過一分鐘。

馬克一號藉由打卡紙讀取、執行每一道指令。它沒有條件分支指令。這表示需要複雜運算的程式碼會很長一串。迴圈的完成需利用打卡紙頭尾相接的方式。這種程式碼與資料分開放置的架構就是眾所皆知的「哈佛



馬克一號左面部份



馬克一號右面部份

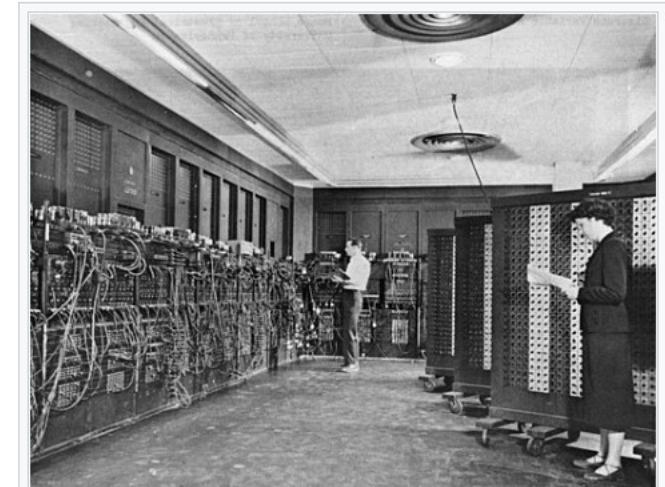


1946 年的通用型電腦 ENIAC

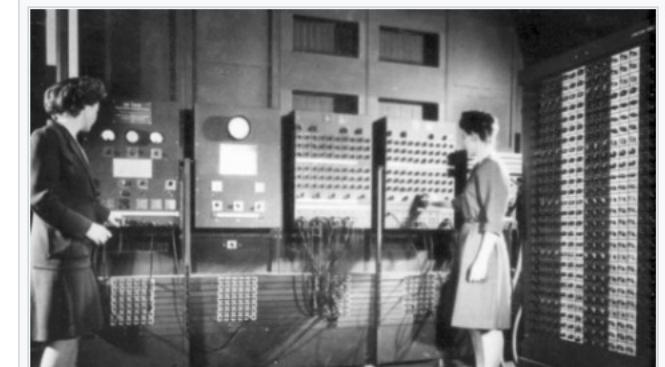
電子數值積分計算機（英語：Electronic Numerical Integrator And Computer），由其縮寫，簡稱為伊尼亞克（英語：ENIAC，發音：*/'enɪ.æk/*，也可稱埃尼阿克）^{[1][2]}是世界上第一台通用計算機。它是圖靈完全的電子計算機，能夠重新編程，解決各種計算問題^[3]。

ENIAC為美國陸軍的彈道研究實驗室（BRL）所使用，用於計算火炮的火力表^{[4][5]}。ENIAC在1946年公布的時候，就被當時的新聞讚譽為「巨腦」。它的計算速度比機電機器提高了一千倍。這是一個飛躍，之前沒有任何一台單獨的機器達到過這個速度。它的數學能力和通用的可程式能力，令當時的科學家和實業家非常激動。發明它的人為了進一步推廣這些新思想，舉辦了一系列關於計算機體系結構的講座。

在二戰期間，美國陸軍資助了ENIAC的設計和建造。建造合同在1943年6月5日簽訂，實際的建造在7月以「PX項目」為代號秘密開始，由賓夕法尼亞大學穆爾電氣工程學院進行。建造完成的機器在1946年2月14日公布^[6]，並於次日在賓夕法尼亞大學正式投入使用^[7]。建造這台機器花費了將近五十萬美元（考慮通貨膨脹，相當於2011年的六百五十萬美元）^[8]。1946年7月，它被美國陸軍軍械兵團正式接受。為了翻新和升級存儲器，ENIAC在1946年11月9日關閉，並在1947年轉移到了馬里蘭州的阿伯丁試驗場。1947年7月，它在那裡重新啟動，繼續工作到1955年10月2日晚上11點45分^[2]。



格倫·貝克（遠）和貝蒂·斯奈德（近）在位於彈道研究實驗室（BRL）Building 328的ENIAC上編程。（美國陸軍照片）



這些電腦

- 雖然大部分不像圖靈機那樣以磁帶為主而是擁有記憶體，然後資料從磁帶讀入不過基本上都和圖靈機具有同樣的能力只是速度會比純粹的磁帶機快很多 ...

馮紐曼

- 曾經使用過 Mark I 和 ENIAC 電腦
並且參與了新電腦 EDVAC 的研發



John von Neumann in the 1940s

他寫了一份 101 頁的報告

First Draft of a Report on the EDVAC

JOHN VON NEUMANN

Introduction

Normally first drafts are neither intended nor suitable for publication. This report is an exception. It is a first draft in the usual sense, but it contains a wealth of information, and it had a pervasive influence when it was first written. Most prominently, Alan Turing cites it, in his proposal for the Pilot ACE,* as the definitive source for understanding the nature and design of a general-purpose digital computer.

After having influenced the first generation of digital computer engineers, the von Neumann report fell out of sight. There were at least two reasons for this: The report was hard to find, and it was very hard to read. This is where its first-draft quality resurfaced. The draft was typed at the Moore School from von Neumann's handwritten manuscript. It is clear that the typescript was never proofread. There are numerous typographical mistakes, and serious misunderstandings about the intended use of mathematical symbols and Greek letters. There are also a considerable number of errors, which may have been in the original manuscript. (Efforts to locate the manuscript have failed, though Herman Goldstine reported that his own archives, now located at Hampshire College, Mass., did contain a copy.)

taken great pains *not* to modify the intended expression, nor to editorialize on the original work. The report is still not easy reading, but to the best of my ability this version is a correct rendering of what von Neumann wrote and intended.

A careful reading of the report will be instructive to anyone with an interest in the past, present, or future of computing.

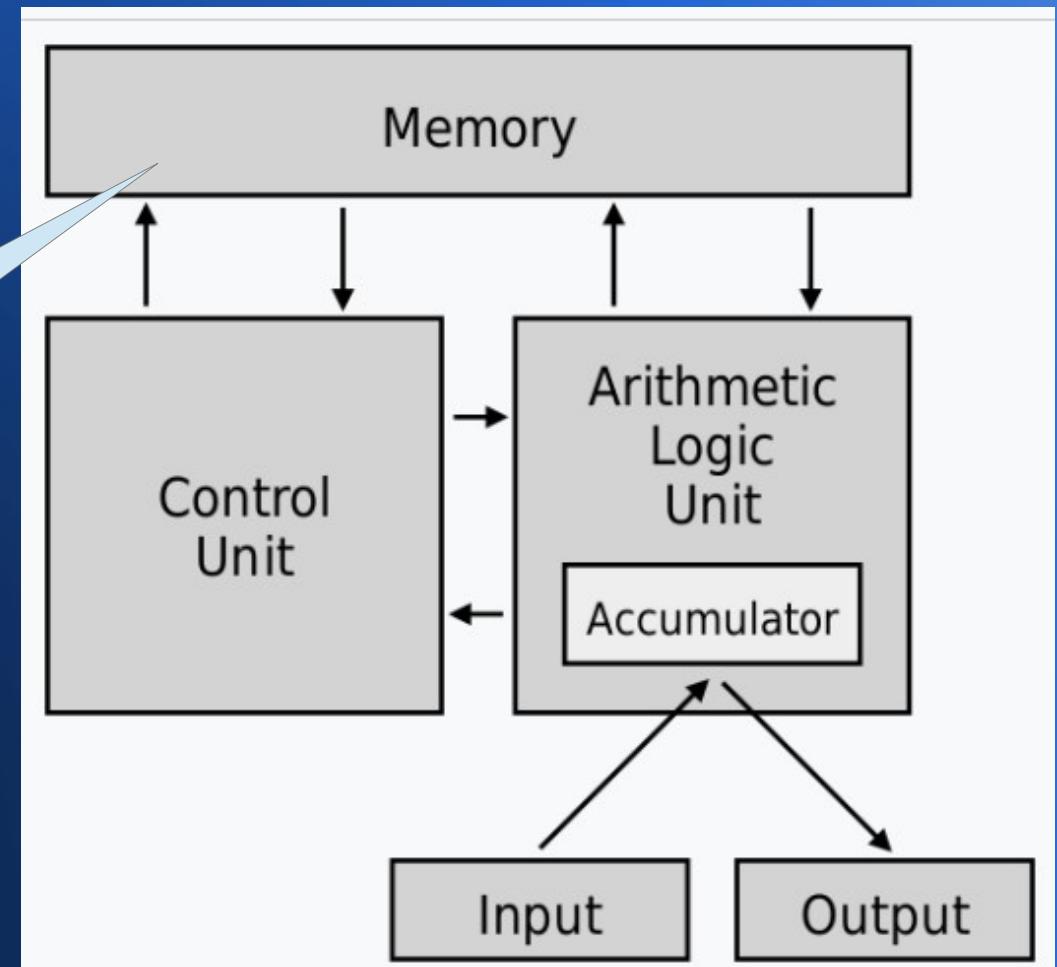
Michael D. Godfrey

該報告

- 提出了預儲程序的觀念
將程式與資料放在同一塊記憶體內
這種架構後來被稱為《馮紐曼架構》

馮紐曼架構

資料和程式都
放在裡面



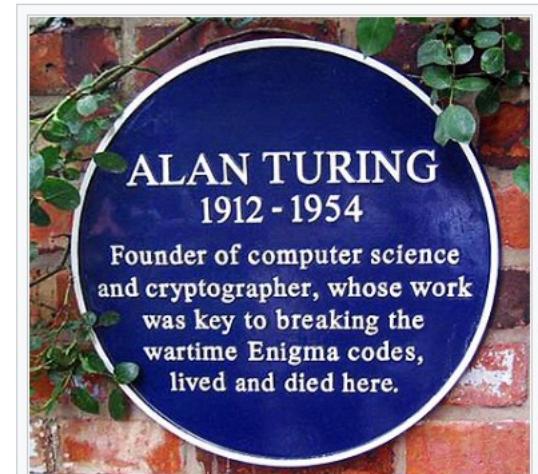
這樣的結構

- 是後來電腦的主流結構

在戰爭結束後

- 圖靈因為同性戀問題而被判化學閹割
後來咬了一口含氰化物的蘋果而死亡

因為圖靈的同性戀傾向而遭到的迫害使得他的職業生涯盡毀。1952年，他的同性伴侶協同一名同謀一起闖進圖靈的房子行竊，但是英國警方的調查結果使得他被控以「明顯的猥褻和性顛倒行為」罪（請參看性悖軌法）。他沒有申辯，並被定罪。在著名的公審後，他被給予了兩個選擇：坐牢或女性荷爾蒙注射「療法」（即化學閹割）。他最後選擇了雌激素注射^[14]，並持續一年。在這段時間裡，藥物產生了包括乳房不斷發育的副作用，也使原本熱愛體育運動的圖靈在身心上受到極大傷害。1954年，圖靈因食用浸過氰化物溶液的蘋果而死亡。很多人相信他有意吃這蘋果，並判決他是自殺。但是他的母親極力爭辯他的死是意外，因為圖靈工作室有很多化學品，而他不小心讓蘋果沾上氰化物溶液。



圖靈在東柴郡威姆斯洛的家，¹⁴掛有藍色牌匾。

但是他對計算機的夢想

- 却持续的不断發展...

計算機

- 變得愈來愈快，愈來愈強大...

同時

- Church 對程式的夢想也繼續發展著

1958 年

- John McCarthy 發明了 LISP 程式語言

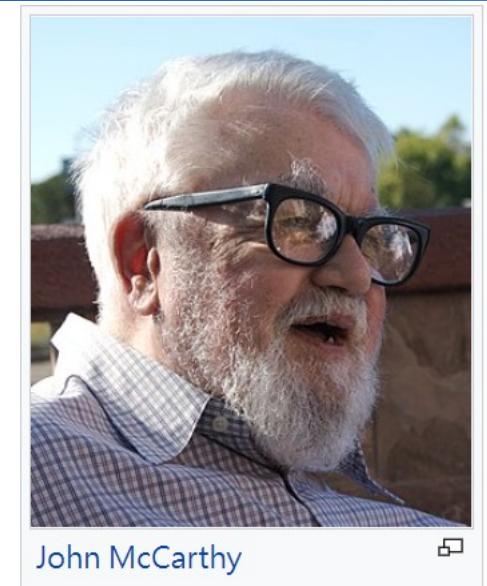
1958年，約翰·麥卡錫在麻省理工學院發明了Lisp程式語言。1960年，他在《ACM通訊》發表論文，名為《遞迴函式的符號表達式以及由機器運算的方式，第一部》^[14]。在這篇論文中闡述了只要透過一些簡單的運算子，以及借鑒自阿隆佐·邱奇的用於匿名函式的表示法，就可以建立一個具圖靈完備性語言，可用於演算法中。

1955年至1956年間，資訊處理語言被創造出來用於人工智慧。它首先使用的列表處理與遞迴概念被用於了Lisp。

麥卡錫最初使用M-表達式寫程式碼，之後再轉成S-表達式，舉例來說M-表達式的語法，`car[cons[A,B]]`，等同於S-表達式的`(car (cons A B))`。然而由於S-表達式具備同像性，即程式與資料由同樣的結構儲存，實際應用中一般只使用S-表達式，而棄用M-表達式。M-表達式曾出現在短暫存在的Horace Enea的MLisp^[15]和Vaughan Pratt的CGOL之中。

約翰·麥卡錫的學生史帝芬·羅素在閱讀完此論文後，認為Lisp程式語言當中的`eval`函式可以用機器碼來實做。他在IBM 704機器上，寫出了第一個Lisp直譯器^[16]。1962年，Tim Hart與Mike Levin在麻省理工學院以Lisp程式語言，實做出第一個完整的Lisp編譯器^[17]。這兩人在筆記中使用的語法比麥卡錫早期的代碼更接近現代Lisp風格。

研究生Daniel Edwards在1962年之前開發的垃圾收集程式，使得在通用計算機上運行Lisp變得實用，但效率仍然是一個問題。在1963年，Timothy Hart提議向Lisp 1.5增加宏^[18]。



John McCarthy



LISP

- 比 λ Calculus 親切得多

```
(defun factorial (n)
  (if (zerop n) 1
      (* n (factorial (1- n)))))
```

```
(defun -reverse (list)
  (let ((return-value))
    (dolist (e list) (push e return-value))
    return-value))
```

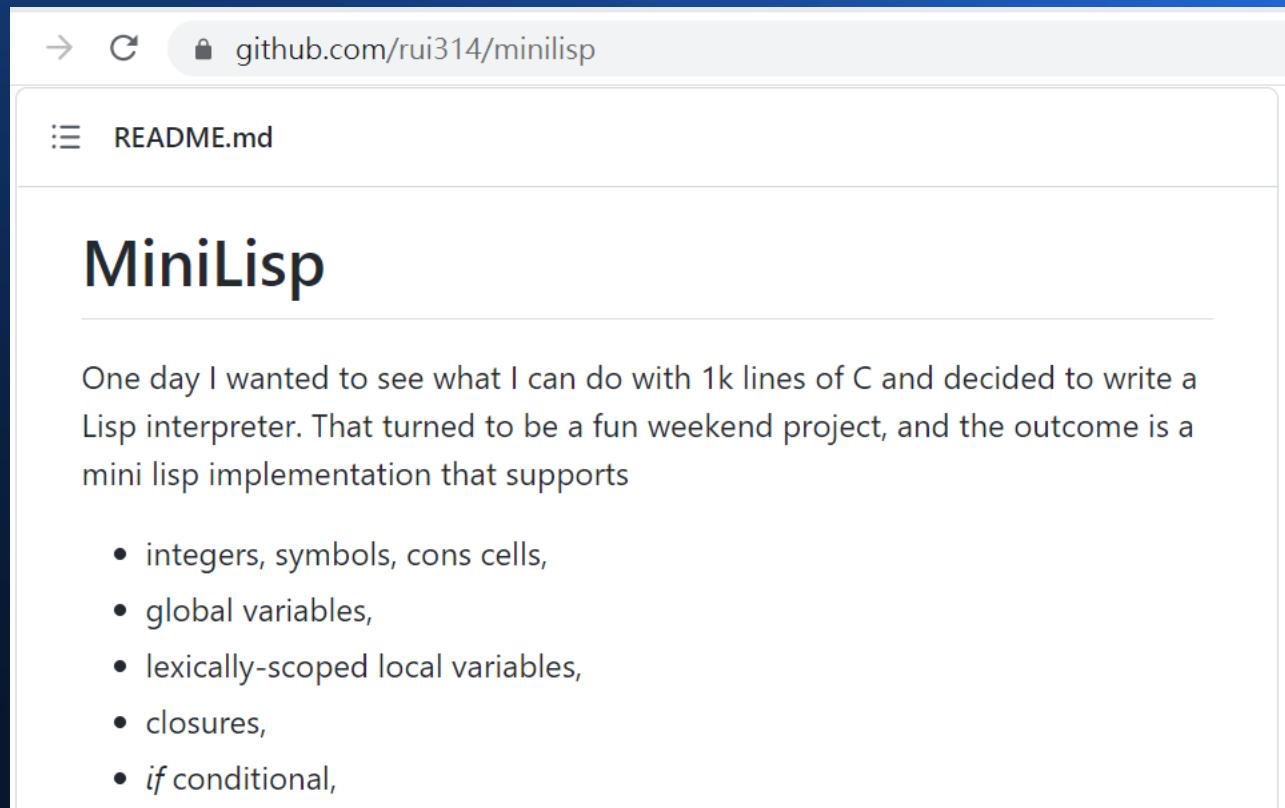
是非常簡潔有力的程式語言

如果你用 C 語言

- 實作 LISP 解譯器，只要幾百行

像是 minilisp 專案

- 就用 996 行 C 語言實現了一個具體而微的 Lisp 解譯器



雖然目前 LISP 的使用者不算很多

- 但卻一直有非常死忠的熱愛者
- Common Lisp 是比較傳統的 LISP 版本
- Scheme 則是 LISP 的一種簡化方言
- Clojure 是現代化在 JVM 平台上跑的 LISP

λ -Calculus

- 啟發了 LISP 的設計

而 LISP

- 又啟發了 Dylan/Julia/Haskell/
OCaml/Erlang/F#/Scala 等語言的設計
- 這類語言通常沒有迴圈，必須使用遞迴
的方式設計
- 被統稱為《函數式程式語言》 ...

後來在 1970 年

- Dennis Ritchie 和 Ken Thompson 合作發展出影響深遠的 C 語言和 UNIX 作業系統
- 讓 C 語言在數十年內成為工業影響力最強的語言

但是

- λ -Calculus 的理論

與 LISP 的簡潔優美

都對電腦與程式領域有深厚的影响

雖然

- 當初希爾伯特想把數學機械化的夢想，被
 - 哥德爾不完備定理
 - Church 的不可解問題證明
 - 圖靈的停止問題證明

所否決了！

但是

- 這些理論的探索，仍然對資訊科學的發展，有著重要的貢獻與長遠的影響
- 也導致了《函數式程式語言》的發明

形成了一條

- 從數理邏輯到 λ -Calculus

再經圖靈機到現代電腦
的發展路徑 ...

而這些知識

- 形成了計算理論的主要架構

串起了從數學經程式到機器的美妙世界

這就是我們今天的

- 十分鐘系列

希望你會喜歡

我們下次見！

Good Bye !