量子计算
—算法篇

Quantum Computer

网址：www.qubits.top

作者：Calvin Tang

邮箱：179209347@qq.com

介绍

本教程包含：

1. 叠加态制备
2. Hadamard Test、SWAP Test
3. 振幅放大
4. 量子傅里叶变换
5. 量子相位估计
6. 量子四则运算：量子加法器、减法器、乘法器，除法器
7. HHL
8. Deutsch-Jozsa
9. Grover搜索

介绍

本教程基于本源量子的Qpanda框架的python版 – **PyQPanda** 编写。

- 一种功能齐全，运行高效的量子软件开发工具包
- QPanda 2是由本源量子开发的开源量子计算框架，它可以用于构建、运行和优化量子算法。
- QPanda 2作为本源量子计算系列软件的基础库，为OriginIR、Qurator、量子计算服务提供核心部件。

QPanda使用文档：
https://pyqpanda-toturial.readthedocs.io/zh/latest/index.html

Github & Gitee 代码地址：
https://github.com/mymagicpower/qubits/tree/main/quantum_algorithm
https://gitee.com/mymagicpower/qubits/tree/main/quantum_algorithm

系统配置和安装

系统配置

pyqpanda是以C++为宿主语言，其对系统的环境要求如下：

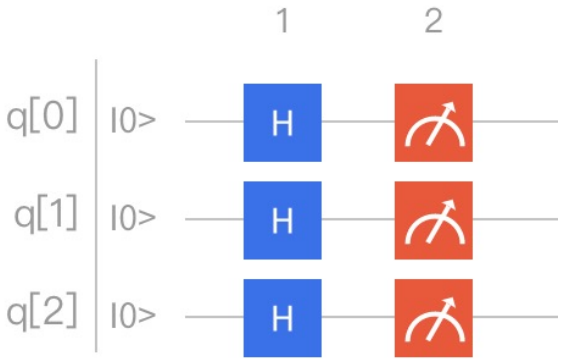
software	version
GCC	>= 5.4.0
Python	>= 3.6.0（建议3.8以支持VQNet）

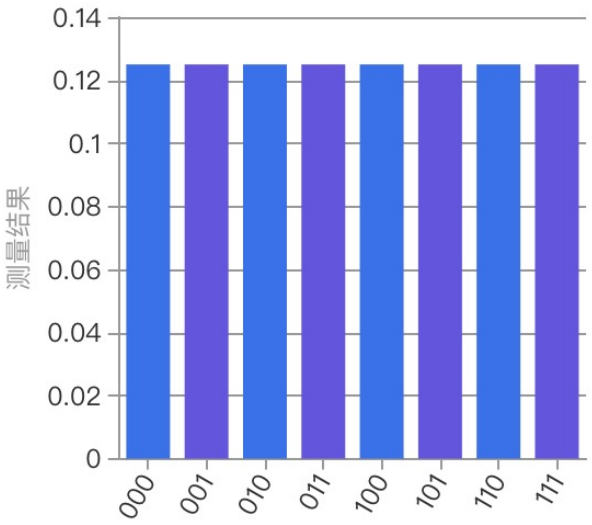
安装 pyqpanda

```
pip install pyqpanda
```

最大叠加态制备 – 3个量子例子

线路设计：





Superposition.py

```
import pyqpanda as pq

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    qubits = machine.qAlloc_many(3)
    prog = pq.create_empty_qprog()

    # 构建量子程序
    prog.insert(pq.H(qubits[0])) \
        .insert(pq.H(qubits[1])) \
        .insert(pq.H(qubits[2]))

    # 对量子程序进行概率测量
    result = pq.prob_run_dict(prog, qubits, -1)
    pq.destroy_quantum_machine(machine)

    # 打印测量结果
    for key in result:
        print(key+":"+str(result[key]))
```

运行结果：

```
000:0.12499999999999735
001:0.12499999999999735
010:0.12499999999999735
011:0.12499999999999735
100:0.12499999999999735
101:0.12499999999999735
110:0.12499999999999735
111:0.12499999999999735
```

Hadamard Test

$$|\psi_3\rangle = |0\rangle \frac{I+U}{2}|\psi\rangle + |1\rangle \frac{I-U}{2}|\psi\rangle$$

$$P_0 = \frac{1+Re(\langle\psi|U|\psi\rangle)}{2},\ P_1 = 1-P_0$$

由公式推导可知，Hadamard Test的结果相应的测量概率均与 $Re(\langle\psi|U|\psi\rangle)$ 即幺正算符 U 在量子态 ψ 上投影期望的实部相关。

代码实例

取 $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$，$U = H$

HadamardTest.py

```
import pyqpanda as pq
if __name__ == "__main__":
    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    cqv = machine.qAlloc_many(1)
    tqv = machine.qAlloc_many(1)
    prog = pq.create_empty_qprog()
    # 构建量子程序
    prog.insert(pq.H(cqv[0])) \
        .insert(pq.H(tqv[0])) \
        .insert(pq.H(tqv[0]).control([cqv[0]]))\
        .insert(pq.H(cqv[0]))
    # 对量子程序进行概率测量
    result = pq.prob_run_dict(prog, cqv, -1)
    pq.destroy_quantum_machine(machine)

    # 打印测量结果
    for key in result:
        print(key+":"+str(result[key]))
```

输出结果应如下所示：

分别以 $\frac{1+\sqrt{2}/2}{2}$ 和 $1-\frac{1+\sqrt{2}/2}{2}$ 的概率得到 $|0\rangle$ 和 $|1\rangle$ 。

```
0:0.8535533905932525
1:0.1464466094067226
```

SWAP Test

代码实例

取 $|\phi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|, |\psi\rangle = |1\rangle$

SWAP Test的一个代码实例如右侧：

输出结果应如下所示，分别以 0.75 和 0.25 的概率得到 |0⟩和 |1⟩ ：

```
0:0.7499999999999841
1:0.2499999999999947
```

SWAPTest.py

```
import pyqpanda as pq
if __name__ == "__main__":
    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    cqv = machine.qAlloc_many(1)
    tqv = machine.qAlloc_many(1)
    qvec = machine.qAlloc_many(1)
    prog = pq.create_empty_qprog()
    # 构建量子程序
        prog.insert(pq.H(cqv[0])) \
       .insert(pq.H(tqv[0])) \
       .insert(pq.X(qvec[0])) \
       .insert(pq.SWAP(tqv[0],qvec[0]).control([cqv[0]]))\
       .insert(pq.H(cqv[0]))

    # 对量子程序进行概率测量
        result = pq.prob_run_dict(prog, cqv, -1)
    pq.destroy_quantum_machine(machine)

    # 打印测量结果
        for key in result:
       print(key+":"+str(result[key]))
```

振幅放大

代码实例 (参考Grover算法，几何解释)

取 $\Omega = \{0,1\}$,

$|\psi\rangle = \cos\frac{\pi}{6}|0\rangle + \sin\frac{\pi}{6}|1\rangle$,

$P_1 = 2|0\rangle\langle 0| - I = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = Z$,

$$P = 2|\psi\rangle\langle\psi| - I = 2\begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{bmatrix}\begin{bmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix} - I = \begin{bmatrix} \frac{1}{2} & \frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & -\frac{1}{2} \end{bmatrix}$$

$$Q = P\,P_1 = \begin{bmatrix} \frac{1}{2} & \frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & -\frac{1}{2} \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix}$$

振幅放大量子线路的相应代码实例如右侧：

$$\theta = \frac{\pi}{3}$$

$$Q = \begin{bmatrix} \cos\frac{\pi}{3} & -sin\frac{\pi}{3} \\ \sin\frac{\pi}{3} & \cos\frac{\pi}{3} \end{bmatrix}$$

AmplitudeAmplification.py

```
import pyqpanda as pq
from numpy import pi

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    qvec = machine.qAlloc_many(1)
    prog = pq.create_empty_qprog()

    # 构建量子程序
     prog.insert(pq.RY(qvec[0], pi/3))
    # prog.insert(pq.Z(qvec[0]))
    prog.insert(pq.RY(qvec[0], pi*2/3))

    # 对量子程序进行概率测量
        result = pq.prob_run_dict(prog, qvec, -1)
    pq.destroy_quantum_machine(machine)

    # 打印测量结果
        for key in result:
       print(key+":"+str(result[key]))
```

输出结果应如下所示，分别以 1 和 0 的概率得到 |0⟩和 |1⟩ ：

```
0:4.930380657631324e-32
1:1.0
```

量子傅里叶变换

QFT在一维情况就是Hadamard量子门。 基于QPanda-2.0的QFT接口函数如下：

```
QFT(qlist)
```

选取 $|x\rangle = |000\rangle$验证QFT的代码实例,输出结果应当以均匀概率 $\frac{1}{8}$ 得到所有量子态，即:

```
000:0.12499999999999735
001:0.12499999999999735
010:0.12499999999999735
011:0.12499999999999735
100:0.12499999999999735
101:0.12499999999999735
110:0.12499999999999735
111:0.12499999999999735
```

QFTDemo.py

```
import pyqpanda as pq
from numpy import pi

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    qvec = machine.qAlloc_many(3)
    prog = pq.create_empty_qprog()

    # 构建量子程序
        prog.insert(pq.QFT(qvec))

    # 对量子程序进行概率测量
        result = pq.prob_run_dict(prog, qvec, -1)
    pq.destroy_quantum_machine(machine)

    # 打印测量结果
        for key in result:
        print(key+":"+str(result[key]))
```

量子相位估计

选取 $U=RY(\frac{\pi}{4}), |\psi\rangle=|0\rangle+i|1\rangle$， 对应的特征值为 $e^{-i\frac{\pi}{8}}$，验证QPE的代码实例如右侧：

```
0000:8.027935818810732e-34
0001:1.4039042059965852e-33
0010:6.324580024821594e-35
0011:1.8002515572777754e-33
0100:1.1716873302045018e-34
0101:3.7185617521916785e-35
0110:7.14676746907725e-35
0111:3.1639374828194345e-33
1000:1.1716873302045018e-34
1001:2.3616836565913952e-33
1010:6.324580024821593e-35
1011:1.5092828476632727e-32
1100:2.019158619146998e-34
1101:2.1419942394604108e-33
1110:1.0410876346494688e-33
1111:0.999999999999936
```

QPEDemo.py

```
import pyqpanda as pq
from numpy import pi
if __name__ == "__main__":
    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    qvec = machine.qAlloc_many(1)
    cqv = machine.qAlloc_many(4)
    prog = pq.create_empty_qprog()
    # 构建量子程序
        prog.insert(pq.H(cqv[0]))\
        .insert(pq.H(cqv[1]))\
        .insert(pq.H(cqv[2]))\
        .insert(pq.H(cqv[3]))\
        .insert(pq.H(qvec[0]))\
        .insert(pq.S(qvec[0]))\
        .insert(pq.RY(qvec[0], pi/4).control(cqv[0]))\
        .insert(pq.RY(qvec[0], pi/2).control(cqv[1]))\
        .insert(pq.RY(qvec[0], pi).control(cqv[2]))\
        .insert(pq.RY(qvec[0], pi*2).control(cqv[3])) \
        .insert(pq.QFT(cqv).dagger())
    # 对量子程序进行概率测量
        result = pq.prob_run_dict(prog, cqv, -1)
    pq.destroy_quantum_machine(machine)
    # 打印测量结果
        for key in result:
        print(key+":"+str(result[key]))
```

量子四则运算 - 量子加法器

在QPanda-2.0中加法器的接口函数如下：

```
QAdder(adder1,adder2,c,is_carry)
QAdderIgnoreCarry(adder1,adder2,c)
QAdd(adder1,adder2,k)
```

前两种接口函数的区别是是否保留进位is_carry，但都只支持正数加法。参数中adder1与adder2为执行加法的比特且格式完全一致，c为辅助比特。

第三种加法器接口函数是带符号的加法器，是基于量子减法器实现的。待加数添加了符号位，相应的辅助比特也从1-2个单比特变为一个adder1.size()+2比特。

加法的输出比特都是adder1，其他非进位比特不变。

量子四则运算 - 量子减法器

基础的加法器只支持非负整数的加法。对于小数要求输入的被加数a和b必须小数点位置相同，小数点对齐后整体长度相同。

对于带符号变换的量子加法，则需要追加辅助比特用于记录符号位。任给两个目标量子态 A,B，对第二个量子态 B 进行特定的补码操作，然后转换为$A-B=A+(-B)$，此处的 $-B$ 并不以符号位取反的方式实现。

该特定的补码操作为：符号位为正则不变，符号位为负需要按位取反后再加1。因此需要一个额外的辅助比特来控制是否进行求补码的操作。

量子减法器实质上就是量子加法器的带符号版本。

在QPanda-2.0中减法器（带符号的加法器）的接口函数如下：

```
QSub(a,b,k)
```

与带符号的加法器相同，两个待减数的量子比特最高位为符号位，辅助比特k.size()=a.size()+2。减法的输出比特是a，其他比特不变。

量子四则运算 - 量子乘法器

量子乘法器是基于加法器完成的。选择乘数 A 作为受控比特，选择乘数 B 以二进制展开逐位作为控制比特，将受控加法器的运算结果累加到辅助比特中。每完成一次 B 控制的受控加法就将乘数 A 左移一位并在末位补零。
于是把通过受控加法输出的数值在辅助比特中累加起来，得到乘法结果。

在QPanda-2.0中乘法器的接口函数如下：

```
QMultiplier(a,b,k,d)
QMul(a,b,k,d)
```

两个接口函数的输入待乘量子比特都包含符号位，但只有QMul支持带符号的乘法运算。

相应的，QMultiplier中，辅助比特k.size()=a.size()+1，结果比特d.size()=2*a.size()。

QMul中，辅助比特k.size()=a.size()，结果比特d.size()=2*a.size()-1。

乘法的输出比特都是d，其他比特不变。

如果等长的输入比特a和b存在小数点，那么在输出比特d中的小数点位置坐标为输入比特中的2倍。

量子四则运算 - 量子除法器

量子除法器是基于量子减法器完成的，通过执行减法后被除数的符号位是否改变来完成大小比较，并决定除法是否终止。除数减去被除数时，商结果加1。每完成一次减法后，重新进行被除数与除数的大小比较，直至除尽或者达到预设精度。因此还需要额外追加一个存储精度参数的辅助比特。

在QPanda-2.0中除法法器的接口函数如下：

```
QDivider(a,b,c,k,t)
QDivider(a,b,c,k,f,s)
QDiv(a,b,c,k,t)
QDiv(a,b,c,k,f,s)
```

与乘法器类似，除法器也是分为两类，尽管输入的待运算比特都带有符号位，但接口分为带符号运算和仅限正数两类。
k为辅助比特，t或s为限制QWhile循环次数的经典比特。
此外，除法器有除不尽的问题，因此可以接口函数有如上四种，对应的输入和输出参数分别有如下性质：
1.QDivider返还余数和商（分别存储在a和c中）时，c.size()=a.size()，但k.size()=a*size()*2+2;
2.QDivider返还精度和商（分别存储在f和c中）时，c.size()=a.size()，但k.size()=3*size()*2+5；
3.QDiv返还余数和商（分别存储在a和c中）时，c.size()=a.size()，但k.size()=a*size()*2+4;
4.QDivider返还精度和商（分别存储在f和c中）时，c.size()=a.size()，但k.size()=a*size()*3+7；
如果参数不能满足量子四则运算所需的比特数目，那么计算依然会进行但结果会溢出。
除法的输出比特是c，带精度的除法中a,b,k都不会变，否则b,k不变但a中存储余数。

代码示例

一个简单的基于QPanda-2.0调用量子四则运算的代码示例：

执行的计算为 (4/1+1−3)∗5=10，因此结果应当以概率 1 得到 |10⟩即 |1010⟩。

```
0001010:1.0
```

QMath.py

```
# 为了节约比特数，辅助比特将会互相借用
 qvm = pq.init_quantum_machine(pq.QMachineType.CPU)
qdivvec = qvm.qAlloc_many(10)
qmulvec = qdivvec[:7]
qsubvec = qmulvec[:-1]
qvec1 = qvm.qAlloc_many(4)
qvec2 = qvm.qAlloc_many(4)
qvec3 = qvm.qAlloc_many(4)
cbit = qvm.cAlloc()
prog = pq.create_empty_qprog()
# (4/1+1-3)*5=10
prog.insert(pq.bind_data(4,qvec3)).insert(pq.bind_data(1,qvec2)) \
  .insert(pq.QDivider(qvec3, qvec2, qvec1, qdivvec, cbit)) \
  .insert(pq.bind_data(1,qvec2)).insert(pq.bind_data(1,qvec2)) \
  .insert(pq.QAdd(qvec1, qvec2, qsubvec)) \
  .insert(pq.bind_data(1,qvec2)).insert(pq.bind_data(3,qvec2)) \
  .insert(pq.QSub(qvec1, qvec2, qsubvec)) \
  .insert(pq.bind_data(3,qvec2)).insert(pq.bind_data(5,qvec2)) \
  .insert(pq.QMul(qvec1, qvec2, qvec3, qmulvec)) \
  .insert(pq.bind_data(5,qvec2))
# 对量子程序进行概率测量
    result = pq.prob_run_dict(prog, qmulvec,1)
pq.destroy_quantum_machine(qvm)
# 打印测量结果
    for key in result:
  print(key+":"+str(result[key]))
```

HHL算法实现

基于QPanda-2.0的HHL算法实现代码较为冗长，此处不作详述，具体参见QPanda-2.0下HHL算法程序源码 ，此处仅介绍QPanda-2.0中提供的几个HHL算法调用接口。

https://github.com/OriginQ/QPanda-2/blob/master/Applications/HHL_Algorithm

```
HHL(matrix, data, QuantumMachine)
HHL_solve_linear_equations(matrix, data)
```

第一个函数接口用于得到HHL算法对应的量子线路，第二个函数接口则可以输入QStat格式的矩阵和右端项，返还解向量。 目前第一个函数接口返回的线路需要追加特殊后处理，得到的并不是直接求解的结果，一般推荐使用第二个函数接口HHL_solve_linear_equations。

HHL算法实现实例

代码实例

取 $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ $b = \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix}$

验证HHL的代码实例如右侧：

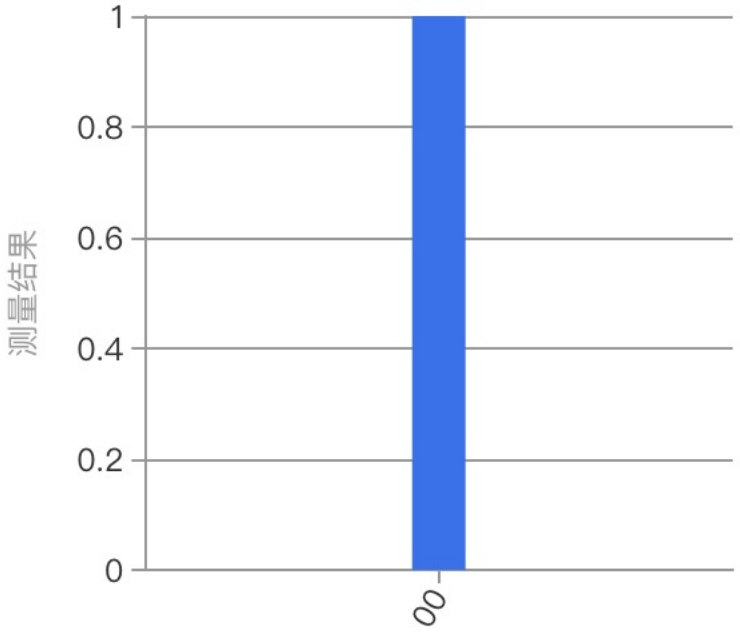
HHLDemo.py

```
import pyqpanda as pq
import numpy as np

if __name__ == "__main__":
  A=[1,0,0,1]
  b=[0.6,0.8]
  result = pq.HHL_solve_linear_equations(A,b,1)

  #打印测量结果
    for key in result:
    print(key)
```

输出结果应该和右端项向量一样是 [0.6,0.8]，虚数项参数为0。因为误差会出现较小的扰动：

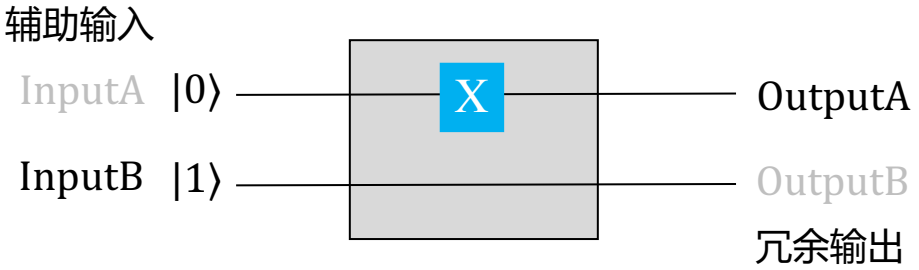
```
(0.5999999999999488+0j)
(0.799999999999932+0j)
```

Deutsch-Jozsa算法 – 2个量子比特线路设计（1/3）



常数函数：

f(x) = |1⟩



测得 $|z_0\rangle = |00\rangle$ 概率为1，该结果说明是常数函数。

Deutsch-Jozsa算法 – 代码实现

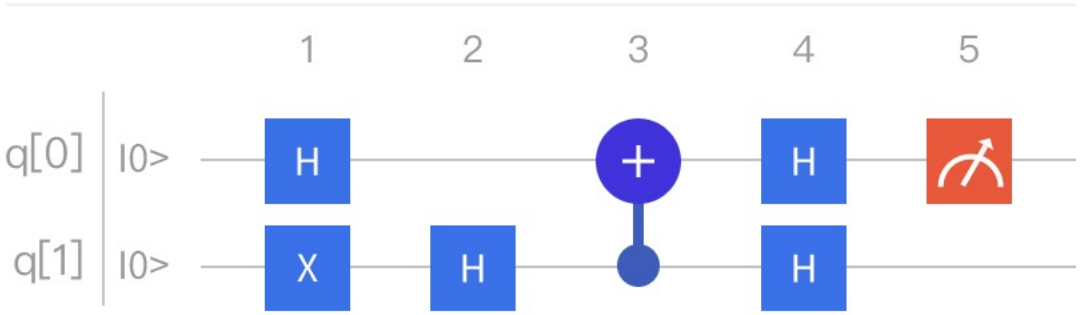
DJ_1.py

```
from pyqpanda import *

if __name__ == "__main__":
    init(QMachineType.CPU)
    qubits = qAlloc_many(2)
    cbits = cAlloc_many(2)
    # 构建量子程序
        prog = QProg()
    prog << H(qubits[0]) << X(qubits[0]) << H(qubits[0])\
        << measure_all(qubits, cbits)

    # 量子程序运行1000次，并返回测量结果
        result = run_with_configuration(prog, cbits, 1000)
    print(result)
    finalize()
```

运行结果：

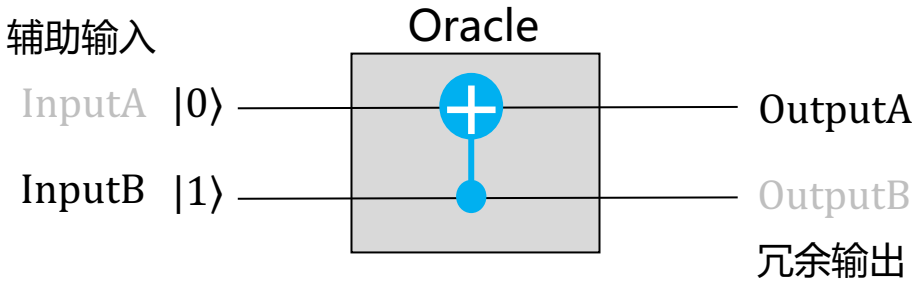
```
{'00': 1000}
```

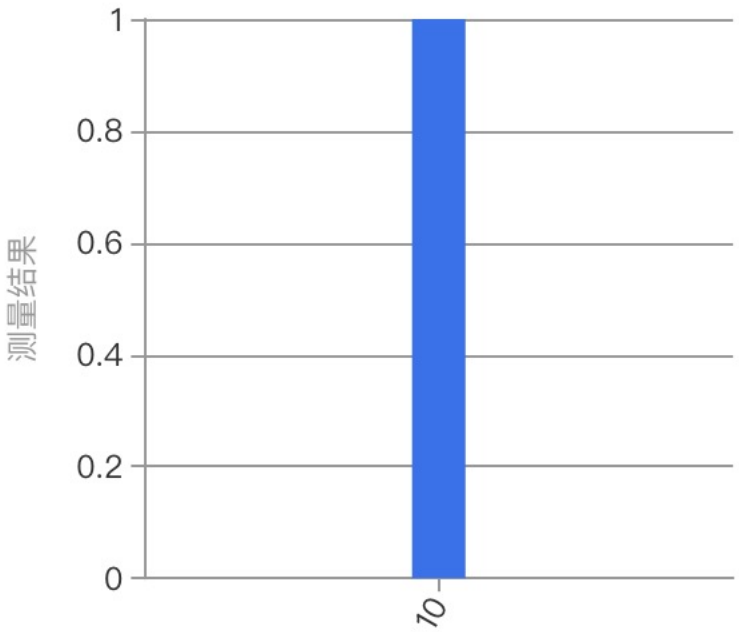
Deutsch-Jozsa算法 – 2个量子比特线路设计（2/3）



f(x) = x

平衡函数：





测得 $|z_1\rangle = |10\rangle$ 概率为1，该结果说明是平衡函数。

Deutsch-Jozsa算法 – 代码实现

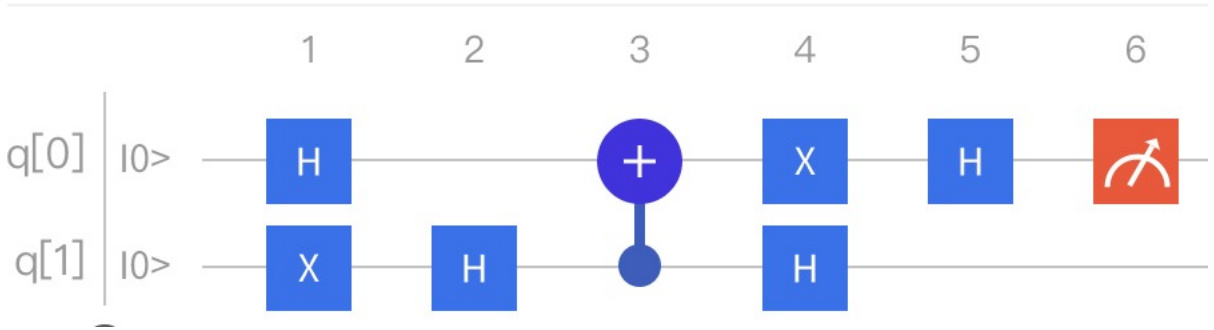
DJ_2.py

```
from pyqpanda import *

if __name__ == "__main__":
    init(QMachineType.CPU)
    qubits = qAlloc_many(2)
    cbits = cAlloc_many(2)
    # 构建量子程序
        prog = QProg()
    prog << H(qubits[0]) \
        << X(qubits[1]) << H(qubits[1]) \
        << CNOT(qubits[1], qubits[0]) \
        << H(qubits[0])\
        << H(qubits[1]) \
        << measure_all(qubits, cbits)

    # 量子程序运行1000次，并返回测量结果
        result = run_with_configuration(prog, cbits, 1000)
    print(result)
    finalize()
```

运行结果：

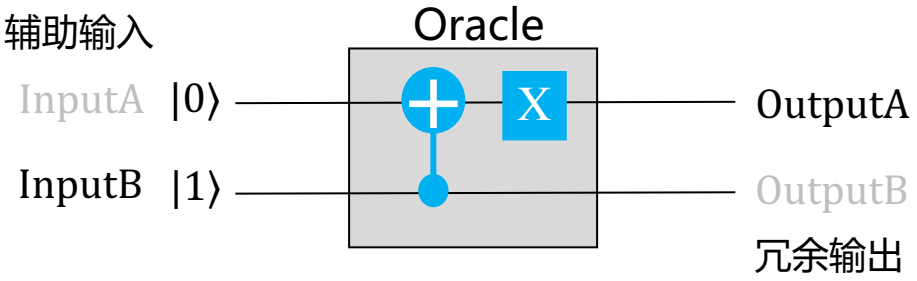
{'10': 1000}

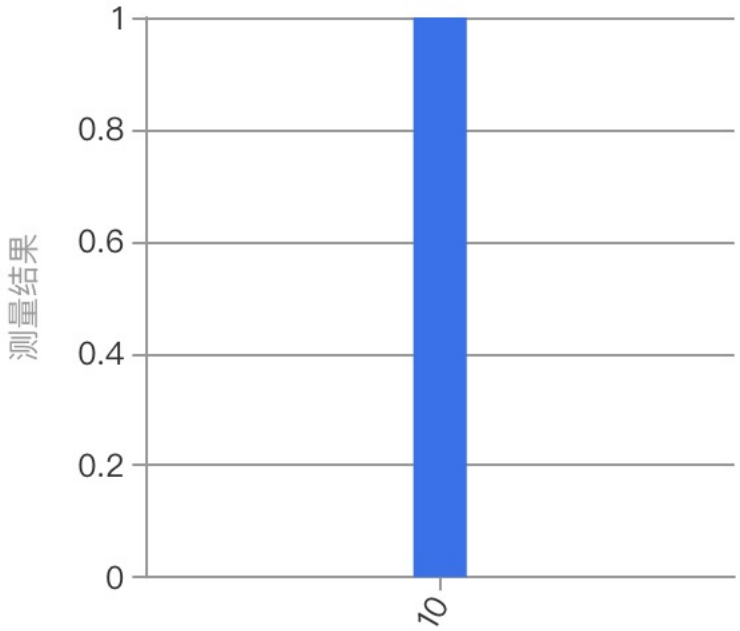
Deutsch-Jozsa算法 – 2个量子比特线路设计（3/3）



f(x) = ¬x

平衡函数：





测得 $|z_1\rangle = |10\rangle$概率为1，该结果说明是平衡函数。

Deutsch-Jozsa算法 – 代码实现

DJ_3.py

```
from pyqpanda import *
import numpy as np
if __name__ == "__main__":
    init(QMachineType.CPU)
    qubits = qAlloc_many(2)
    cbits = cAlloc_many(2)
    # 构建量子程序
        prog = QProg()
    prog << H(qubits[0]) \
        << X(qubits[1])<< H(qubits[1]) \
        << CNOT(qubits[1], qubits[0]) \
        << X(qubits[0]) << H(qubits[0])\
        << H(qubits[1]) \
        << measure_all(qubits, cbits)

    # 量子程序运行1000次，并返回测量结果
        result = run_with_configuration(prog, cbits, 1000)
    print(result)
    finalize()
```

运行结果：

{'10': 1000}

Grover 算法代码实现例子 - 1

Grover_1.py

```
from pyqpanda import *

if __name__ == "__main__":
    machine = CPUQVM()
    machine.initQVM()
    x = machine.cAlloc()
    data = [0,3,2,1]
    measure_qubits = QVec()

    # 构建Grover算法量子线路
        grover_cir = Grover(data, x==2, machine, measure_qubits, 1)
    cbits = machine.cAlloc_many(len(measure_qubits))
    prog = QProg()
    prog << grover_cir << measure_all(measure_qubits, cbits)

    # 量子程序运行1000次，并返回测量结果
        result = machine.run_with_configuration(prog, cbits, 1000)
    print(result)
    finalize()
```

运行结果：

{'10': 1000}

Grover 算法代码实现例子 - 2

Grover_2.py

```
#!/usr/bin/env python

import pyqpanda as pq
import numpy as np

if __name__ == "__main__":

  machine = pq.init_quantum_machine(pq.QMachineType.CPU)
  x = machine.cAlloc()
  prog = pq.create_empty_qprog()

  data=[3, 6, 6, 9, 10, 15, 11, 6]
  grover_result = pq.Grover_search(data, x==6, machine, 1)

  print(grover_result[1])
```

运行结果：

```
[1, 2, 7]
```

Thank You