

---

# QPanda2 Introduction

OriginQC

Aug 17, 2019



# Contents

|          |                             |           |
|----------|-----------------------------|-----------|
| <b>1</b> | <b>写在前面</b>                 | <b>3</b>  |
| 1.1      | 0.1 量子                      | 3         |
| 1.2      | 0.2 量子计算                    | 3         |
| 1.3      | 0.3 量子经典混合编程                | 3         |
| <b>2</b> | <b>第 1 章 Quick Start</b>    | <b>5</b>  |
| 2.1      | 1.1 QRunes 简介               | 5         |
| 2.2      | 1.2 QRunes 开发环境             | 5         |
| 2.3      | 1.3 Qurator 介绍              | 9         |
| 2.4      | 1.4 初窥 QRunes               | 9         |
| <b>3</b> | <b>第 2 章 QRunes 基本类型与变量</b> | <b>13</b> |
| 3.1      | 2.1 QRunes 的基本对象类型          | 13        |
| 3.2      | 2.2 字面值常量                   | 18        |
| 3.3      | 2.2.1 整型字面值规则               | 18        |
| 3.4      | 2.2.2 浮点字面值规则               | 19        |
| 3.5      | 2.2.3 布尔字面值                 | 19        |
| 3.6      | 2.3 变量                      | 19        |
| 3.7      | 2.3.1 什么是变量                 | 19        |
| 3.8      | 2.3.2 变量名                   | 20        |
| 3.9      | 2.3.3 关键字                   | 20        |
| 3.10     | 2.3.4 变量命名习惯                | 20        |
| 3.11     | 2.3.5 变量的定义                 | 21        |
| <b>4</b> | <b>第 3 章 QRunes 表达式</b>     | <b>23</b> |
| 4.1      | 3.1 操作符                     | 24        |
| 4.2      | 3.1.1 一元操作符                 | 24        |
| 4.3      | 3.1.2 二元操作符                 | 24        |
| 4.4      | 3.1.3 三元操作符                 | 26        |
| 4.5      | 3.1.4 逗号操作符                 | 26        |
| 4.6      | 3.1.5 操作符详解                 | 26        |
| 4.7      | 3.2 表达式                     | 28        |
| 4.8      | 3.2.1 主表达式                  | 28        |

|          |                                     |            |
|----------|-------------------------------------|------------|
| 4.9      | 3.2.2 括号表达式 . . . . .               | 29         |
| 4.10     | 3.2.3 复合表达式 . . . . .               | 29         |
| 4.11     | 3.2.4 常量表达式 . . . . .               | 30         |
| 4.12     | 3.2.5 lambda 表达式 . . . . .          | 30         |
| <b>5</b> | <b>第 4 章语句</b>                      | <b>33</b>  |
| 5.1      | 4.1 简单语句 . . . . .                  | 33         |
| 5.2      | 4.2 复合语句 . . . . .                  | 34         |
| 5.3      | 4.3 函数调用语句 . . . . .                | 34         |
| 5.4      | 4.4 辅助类型控制语句 . . . . .              | 36         |
| 5.5      | 4.5 量子类型控制语句 . . . . .              | 38         |
| 5.6      | 4.6 量子比特测量语句 . . . . .              | 39         |
| 5.7      | 4.7 return 语句 . . . . .             | 39         |
| <b>6</b> | <b>第 5 章函数</b>                      | <b>41</b>  |
| 6.1      | 5.1 函数的声明 . . . . .                 | 41         |
| 6.2      | 5.2 函数的定义 . . . . .                 | 41         |
| 6.3      | 5.3 函数的参数 . . . . .                 | 42         |
| 6.4      | 5.4 函数的返回值 . . . . .                | 42         |
| 6.5      | 5.5 函数调用 . . . . .                  | 45         |
| <b>7</b> | <b>第 6 章常用量子算法介绍及实现</b>             | <b>47</b>  |
| 7.1      | 6.1 QAOA 算法 . . . . .               | 47         |
| 7.2      | 6.2 Deutsch-Jozsa 算法 . . . . .      | 57         |
| 7.3      | 6.3 Grover 算法 . . . . .             | 62         |
| 7.4      | 6.4 HHL 算法 . . . . .                | 71         |
| 7.5      | 6.5 QuantumWalk 算法 . . . . .        | 76         |
| 7.6      | 6.6 Simon 算法 . . . . .              | 80         |
| 7.7      | 6.7 CoinFlip 算法 . . . . .           | 88         |
| 7.8      | 6.8 Bernstein-Vazirani 算法 . . . . . | 96         |
| 7.9      | 6.9 QPE 算法 . . . . .                | 104        |
| 7.10     | 6.10 Shor 算法 . . . . .              | 111        |
| <b>8</b> | <b>第 7 章 QRunes 类型系统</b>            | <b>125</b> |
| 8.1      | 7.1 量子类型 Quantum Type . . . . .     | 125        |
| 8.2      | 7.2 辅助类型 Auxiliary Type . . . . .   | 126        |
| 8.3      | 7.3 经典类型 Classical Type . . . . .   | 127        |

目录:



# Chapter 1

## 写在前面

### 1.1 0.1 量子

量子 (quantum) 是现代物理的重要概念。最早是由德国物理学家 M·普朗克在 1900 年提出的。他假设黑体辐射中的辐射能量是不连续的, 只能取能量基本单位的整数倍, 从而很好地解释了黑体辐射的实验现象。

后来的研究表明, 不但能量表现出这种不连续的分离化性质, 其他物理量诸如角动量、自旋、电荷等也都表现出这种不连续的量子化现象。这同以牛顿力学为代表的经典物理有根本的区别。量子化现象主要表现在微观物理世界。描写微观物理世界的物理理论是量子力学。

量子一词来自拉丁语 quantus, 意为“有多少”, 代表“相当数量的某物质”。自从普朗克提出量子这一概念以来, 经爱因斯坦、玻尔、德布罗意、海森伯、薛定谔、狄拉克、玻恩等人的完善, 在 20 世纪的前半期, 初步建立了完整的量子力学理论。绝大多数物理学家将量子力学视为理解和描述自然的基本理论。

### 1.2 0.2 量子计算

量子计算是一种遵循量子力学规律调控量子信息单元进行计算的新型计算模式。对照于传统的通用计算机, 其理论模型是通用图灵机; 通用的量子计算机, 其理论模型是用量子力学规律重新诠释的通用图灵机。从可计算的问题来看, 量子计算机只能解决传统计算机所能解决的问题, 但是从计算的效率上, 由于量子力学叠加性的存在, 目前某些已知的量子算法在处理问题时速度要快于传统的通用计算机。

### 1.3 0.3 量子经典混合编程

QRunes 根据量子计算的经典与量子混合 (Quantum-Classical Hybrid) 特性, 在程序编译之后可以操纵宿主机、量子测控设备与量子芯片来实现量子计算。所谓的量子经典混合编程即是将实现量子计算的部分用量子代码实现量子比特和量子逻辑门的操作, 而与经典计算机打交道的部分 (结果展示、量子程序开发包, 比如 QPanda) 用经典语言实现的一种编程方式成为量子经典混合编程。





## Chapter 2

# 第 1 章 Quick Start

### 2.1 1.1 QRunes 简介

QRunes 是一种面向过程、命令式的量子编程语言 Imperative language（这也是当前主流的一种编程范式），它的出现是为了实现量子算法。QRunes 根据量子计算的经典与量子混合（Quantum-Classical Hybrid）特性，在程序编译之后可以操纵经典计算机与量子芯片来实现量子计算。

QRunes 通过提供高级语言的形式（类 C 的编程风格）来量子算法的实现和程序逻辑控制。其丰富的类型系统 (Quantum Type, Auxiliary Type, Classical Type) 可以实现量子计算中数据对象的绑定和行为控制，可以满足各类量子算法开发人员的算法实现需求。

QRunes 构成：Settings, QCodes 和 Script。其中 Settings 部分定义了关于 QRunes 编译的全局信息；QCodes 部分是具体的对于量子比特操作和行为的控制；Script 部分是宿主程序的实现，它的实现依赖于经典编程语言（C++，Python 等）和相关联的量子程序开发工具包（比如：QPanda/pyQPanda）。

### 2.2 1.2 QRunes 开发环境

#### 2.2.1 1.2.1 QRunes 与 QPanda/pyQPanda

##### #### QPanda

QPanda SDK 是由本源量子推出的，基于量子云服务的，开源的量子软件开发包。用户可基于此开发包开发在云端执行的量子程序。QPanda 使用 C++ 语言作为经典宿主语言，支持以 QRunes 书写的量子语言。目前，QPanda 支持本地仿真运行模式和云仿真运行模式，最高可支持到 32 位。Q-Panda 提供了一个可执行的命令行程序，通过指令控制量子程序的加载、运行和读出。另外，QPanda 提供了一组 API，可供用户自行定制功能。

##### #### QPanda 2

QPanda 2(Quantum Programming Architecture for NISQ Device Applications) 是一个高效的量子计算开发工具库，可用于实现各种量子算法，QPanda 2 基于 C++ 实现，并可扩展到 Python。

### #### PyQPanda

PyQPanda 是我们通过 pybind11 工具，以一种直接和简明的方式，对 QPanda2 中的函数、类进行封装，并且提供了几乎完美的映射功能。封装部分的代码在 QPanda2 编译时会生成为动态库，从而可以作为 python 的包引入。

## 2.2.2 1.2.2 开发环境配置与运行

为了兼容高效与便捷，我们为您提供了 C++ 和 Python (pyQPanda) 两个版本，pyQPanda 封装了 C++ 对外提供的接口。

### #### C++ 的使用

使用 QPanda 2 相对于 pyQPanda 会复杂一些，不过学会编译和使用 QPanda 2，您会有更多的体验，更多详情可以阅读 [使用文档](#)。话不多说，我们先从介绍 Linux 下的编译环境开始。

### #### 编译环境

在下载编译之前，我们需要：

| software | version      |
|----------|--------------|
| GCC      | $\geq 5.4.0$ |
| CMake    | $\geq 3.1$   |
| Python   | $\geq 3.6.0$ |

### #### 下载和编译

我们需要在 Linux 终端下输入以下命令：

- `$ git clone https://github.com/OriginQ/QPanda-2.git`
- `$ cd qpanda-2`
- `$ mkdir build`
- `$ cd build`
- `$ cmake -DCMAKE_INSTALL_PREFIX=/usr/local ..`
- `$ make`

### #### 安装

编译完成后，安装就简单的多，只需要输入以下命令：

- `$ make install`

## #### 开始量子编程

现在我们来到最后一关，创建和编译自己的量子应用。我相信对于关心如何使用 QPanda 2 的朋友来说，如何创建 C++ 项目，不需要我多说。不过，我还是需要提供 CMakeList 的示例，方便大家参考。

```
cmake_minimum_required(VERSION 3.1)
project(testQPanda)
SET(CMAKE_INSTALL_PREFIX "/usr/local")
SET(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_INSTALL_PREFIX} lib/cmake")

add_definitions("-std=c++14 -w -DGTEST_USE_OWN_TR1_TUPLE=1")
set(CMAKE_BUILD_TYPE "Release")
set(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -g -ggdb")
set(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3")
add_compile_options(-fPIC -fpermissive)
find_package(QPANDA REQUIRED)
if (QPANDA_FOUND)

    include_directories(${QPANDA_INCLUDE_DIR}
                        ${THIRD_INCLUDE_DIR})
    add_executable(${PROJECT_NAME} test.cpp)
    target_link_libraries(${PROJECT_NAME} ${QPANDA_LIBRARIES})
endif (QPANDA_FOUND)
```

我们接下来通过一个示例介绍 QPanda 2 的使用，此例子构造了一个量子叠加态。在量子程序中依次添加 H 门和 CNOT 门，最后对所有的量子比特进行测量操作。此时，将有 50% 的概率得到 00 或者 11 的测量结果。

```
#include "QPanda.h"
#include <stdio.h>
using namespace QPanda;
int main()
{
    init(QMachineType::CPU);
    QProg prog;
    auto q = qAllocMany(2);
    auto c = cAllocMany(2);
    prog << H(q[0])
        << CNOT(q[0],q[1])
        << MeasureAll(q, c);
    auto results = runWithConfiguration(prog, c, 1000);
    for (auto result : results){
        printf("%s : %d\n", result.first.c_str(), result.second);
    }
```

(continues on next page)

(continued from previous page)

```
}  
    finalize();  
}
```

最后，编译，齐活。

```
$ mkdir build  
$ cd build  
$ cmake ..  
$ make
```

运行结果如下：

```
00 : 512  
11 : 488
```

### #### python 的使用

pyQPanda 只需要通过 pip 就可安装使用。

- -pip install pyqpanda

我们接下来通过一个示例介绍 pyQPanda 的使用，此例子构造了一个量子叠加态。在量子程序中依次添加 H 门和 CNOT 门，最后对所有的量子比特进行测量操作。此时，将有 50% 的概率得到 00 或者 11 的测量结果。

```
from pyqpanda import *  
  
init(QMachineType.CPU)  
prog = QProg()  
q = qAlloc_many(2)  
c = cAlloc_many(2)  
prog.insert(H(q[0]))  
prog.insert(CNOT(q[0],q[1]))  
prog.insert(measure_all(q,c))  
result = run_with_configuration(prog, cbit_list = c, shots = 1000)  
print(result)  
finalize()
```

运行结果如下：

```
{'00': 493, '11': 507}
```

## 2.3 1.3 Qurator 介绍

qurator-vscode 是本源量子推出的一款可以开发量子程序的 VS Code 插件。其支持 QRunes2 语言量子程序开发，并支持 Python 和 C++ 语言作为经典宿主语言。

在 qurator-vscode 中，量子程序的开发主要分为编写和运行两个部分。

- 编写程序：插件支持模块化编程，在不同的模块实现不同的功能，其中量子程序的编写主要在 qcodes 模块中；
- 程序运行：即是收集结果的过程，插件支持图表化数据展示，将运行结果更加清晰的展现在您的面前。

### 2.3.1 1.3.1 qurator-vscode 设计思想

考虑到目前量子程序的开发离不开经典宿主语言的辅助，qurator-vscode 插件设计时考虑到以下几点：

1. 模块编程：qurator-vscode 插件支持模块编程，将整体程序分为三个模块：settings、qcodes 和 script 模块。在不同的模块完成不同的功能。在 settings 模块中，您可以进行宿主语言类型、编译还是运行等设置；在 qcodes 模块中，您可以编写 QRunes2 语言程序；在 script 模块中，您可以编写相应的宿主语言程序。
2. 切换简单：qurator-vscode 插件目前支持两种宿主语言，分别为 Python 和 C++。您可以在两种宿主语言之间自由的切换，您只需要在 settings 模块中设置 language 的类型，就可以在 script 模块中编写对应宿主语言的代码。插件会自动识别您所选择的宿主语言，并在 script 模块中提供相应的辅助功能。
3. 图形展示：qurator-vscode 插件提供图形化的结果展示，程序运行后会展示 json 格式的运行结果，您可以点击运行结果，会生成相应的柱状图，方便您对运行结果的分析。

## 2.4 1.4 初窥 QRunes

### 2.4.1 1.4.1 QRunes 关键字

|        |                    |
|--------|--------------------|
| int    | Hamiltonian        |
| float  | variationalCircuit |
| double | var                |
| bool   | circuitGen         |
| map    |                    |
| qubit  |                    |
| cbit   |                    |
| vector |                    |

## 2.4.2 1.4.2 QRunes 程序结构

### ### QRunes 由三部分组成

- ##### settings 模块中可以设置宿主语言，编译还是运行；

```
@settings:
    language = Python;
    autoimport = True;
    compile_only = False;
```

- ##### qcodes 模块中可以编写 QRunes2 量子语言代码；

```
D_J(qvec q,cvec c){
    RX(q[1],Pi);
    H(q[0]);
    H(q[1]);
    CNOT(q[0],q[1]);
    H(q[0]);
    Measure(q[0],c[0]);
}
```

- ##### script 模块中可以编写宿主语言代码，目前支持 Python 和 C++ 两种宿主语言。

```
init(QuantumMachine_type.CPU_SINGLE_THREAD)
q = qAlloc_many(2)
c = cAlloc_many(2)
qprog1 = D_J(q,c)
result = directly_run(qprog1)
print(result)
finalize()
```

## 2.4.3 1.4.3 Oops! 你的第一个量子程序

点击右上方 Run this QRunes 运行程序，或者使用命令提示符 qurator-vscode: Run this QRunes 来运行程序 (快捷键 F5)，点击运行结果可以以柱状图的形式展示。

```

1  @settings:
2      language = Python;
3      autoimport = True;
4      compile_only = False;
5  @qcodes:
6      // The D_1 Algorithm of 2 quantum bits
7  D_1(qvec q,cvec c){
8      RX(q[1],Pi);
9      H(q[0]);
10     H(q[1]);
11     CNOT(q[0],q[1]);
12     H(q[0]);
13     Measure(q[0],c[0]);
14 }
15 @script:
16 init(QMachineType.CPU_SINGLE_THREAD)
17 q = qAlloc_many(2)
18 c = cAlloc_many(2)
19 qprog1 = D_1(q,c)
20 result = directly_run(qprog1)
21 print(result)
22 finalize()
23


```

**Result:**

{c0: True}

---

**Click data to show chart:**



##### 小结





## Chapter 3

# 第 2 章 QRunes 基本类型与变量

### 3.1 2.1 QRunes 的基本对象类型

#### 3.1.1 2.1.1 量子类型 Quantum Type

量子类型的对象描述的是量子芯片上的量子比特。

例如: qubit q;

它在运行期间会映射某一个量子芯片上的量子比特。这种对象只能被最终用于量子逻辑门操作中。它的赋值相当于创建这个映射的别名, 并非描述其中的数据的复制。

#### 3.1.2 2.1.2 量子线路类型 Quantum Circuit Type

量子线路, 也称量子逻辑电路是最常用的通用量子计算模型, 表示在抽象概念下, 对于量子比特进行操作的线路, 是各种逻辑门组成的集合。最后常需要量子测量将结果读取出来。不同于传统电路是用金属线所连接以传递电压讯号或电流讯号, 在量子线路中, 线路是由时间所连接, 亦即量子比特的状态随着时间自然演化, 过程中是按照哈密顿运算符的指示, 一直到遇上逻辑门而被操作。由于组成量子线路的每一个量子逻辑门都是一个酉矩阵, 所以整个量子线路整体也是一个大的酉矩阵。首先, 初始化一个量子线路对象有以下两种风格:

C++ 风格

```
circuit cir;
```

C 语言风格 (主要有两种方式)

```
//方式 1  
circuit cir = Createqc(); //Createqc is the initialization method()
```

(continues on next page)

(continued from previous page)

```
//方式 2
circuit qc2 = qc1; //qc1 has already defined
```

dagger 的作用是复制一份当前的量子线路，并更新复制的量子线路的 dagger 标记。举个例子：

```
circuit cir;
circuit cir_dagger = cir.dagger();
```

除了转置共轭操作，您也可以为量子线路添加控制比特。circuit 类型为您内置了两个成员函数用于添加控制比特：control、setControl。

control 的作用是复制当前的量子线路，并给复制的量子线路添加控制比特，例如：

```
circuit cir;
circuit cir_control = cir.control(qvec);
```

上述都需要接收一个参数，参数类型为 QVec，QVec 是 qubit 的 vector 容器类型。我们将在下一部分详细介绍。

### 3.1.3 2.1.3 数组类型 Array Construct Type

#### 2.1.3.1 类型声明

vector 是同一种类型的对象的集合，每个对象都有一个对应的整数索引值（对象和变量的区分和 C++ 类似）。在 QRunes 中，系统将负责管理与存储元素相关的内存。和 C++ 一样，因为它可以包含其他对象，我们也可以将这一类型称为容器。必须明确的是，一个容器中的所有对象都必须是同一类型的。在 QRunes 中，容器的内置类型只支持量子类型、辅助类型和经典类型。使用容器可以写一个声明或函数定义，从而用于多个不同的数据类型。因此，我们可以定义保存经典类型对象的容器，或保存辅助类型对象的容器。

声明数组类型的对象，需要提供附加信息。详细的说，就是要提供具体的类型，即容器保存的是何种对象的类型，通过将类型放在名称后面的尖括号中来指定类型，如：

```
vector <int> ives;      // ivec holds objects of type auxiliary primary type
vector <cbit> cvec;    // cvec holds objects of classical primary type
```

和 C++ 类似，定义数组类型对象需要指定类型和一个变量的列表。上面的第一个定义，类型是 vector <int>，该类型即是含有若干 int 型对象的 vector，变量名为 ivec。第二个定义的变量名是 cvec，它所保存的元素都是经典类型的对象。我们需要注意的是，数组类型为我们提供了一套类似于 C++ 中的类模板机制，我们可以用它来定义多种数据类型。数组类型的内置类型的每一种都指定了其保存元素的类型。因此，我们可以简单地认为上述两个例子都是一种数据类型。

#### 2.1.3.2 数组类型的操作

切片（slice）是对数组一个连续片段的引用，因此切片是一个引用类型。切片的内部结构包含开始位置地址（&）、大小（length）和容量（cap）。切片并不存储任何数据，它只是描述了底层数组中的一段。更改切片的元

素会修改其底层数组中对应的元素。

切片的使用和 Python 类似，由数组生成新的切片主要通过两个下标来界定，即一个上界和一个下界，二者以冒号分隔，它的一般形式为：

```
a[low : high]
```

以下为切片的内部结构的详细含义

地址：即切片创建时指向一个底层数组元素的指针

长度：即切片内部元素数量，可以用 length 求得

容量：当长度大于容量时，成倍增长

在 QRunes 中，切片的上界和下界必须指定，不允许使用默认值操作。不难看出，从数组生成新的切片拥有以下特性：

- (1) 取出的元素数量为：结束位置-开始位置
- (2) 取出元素不包含结束位置对应的索引，切片最后一个元素使用 slice[length(slice)] 获取
- (3) 上界下界同时为 0 时，等效于空切片，一般用于切片复位

### 3.1.4 2.1.4 辅助类型 Auxiliary Type

辅助类型是为了更方便创建量子操作的辅助对象。它在编译后的程序中不存在。它可以用于描述一些用于决定量子程序构造的变量或者常量。它也可以是一个编译期间的 if 判断或者 for 循环。

对于一组 qubit，例如 vector<qubit> qs，我们要创建作用在它们上面的 Hadamard 门，我们可以利用如下语句：

```
for (i = 0; i < qs.length()) {
    H(qs[i]);
}
```

这一组语句是一个典型的 for 循环，但是执行这个程序的时机是在编译期间，因此这个 for 循环并不是在量子计算机中运行的 for 循环。它的效果相当于全部展开。即：

```
H(q[0]);
H(q[1]);
H(q[2]);
...
```

### 3.1.5 2.1.5 经典类型 Classical Type

经典类型是在量子测控系统中存在的对象。他们的创建、计算、管理都是由量子芯片的测控系统完成的。这个系统具有实时性的特点，因此这些变量的生命周期和 qubit 的退相干时间共存。它是为了解决普通的宿主主机和量子芯片之间无法进行实时数据交换的问题而存在的。简而言之，它们介于宿主机（辅助类型）和量子芯片（量子类型）之间。

经典类型的变量典型地可以被用于保存量子比特的测量结果。除此之外，由测量结果决定的 IF 和 WHILE 操作，即后面会提到的 QIF, QWHILE 操作也是在测控系统中完成的，所以也属于经典类型。要注意到 QIF 和 QWHILE 和宿主机（辅助类型）的 if, for, while 等操作具有完全不同的运行时机，其中辅助类型的变量、表达式、语句等是编译期间计算的，经典类型是运行期间计算的。

例如：

```
cbit c;
qubit q;
H(q);
measure(q,c);
qif(c){
    // do something...
}
```

这个程序就根据一个 qubit 在执行完 Hadamard 门之后进行的测量的结果来选择执行分支。注意到 c 是一个在测控系统中存在的变量，而 qif 的判断也是在这个系统中实时完成的，之间与宿主机不会发生数据传输。

经典变量之间还可以进行计算，比如：

```
qif(!c) {} // 对 c 求非
qif(c1 == c2) {} //比较 c1 与 c2 的值
qif(c1 == True) {} //等价于 qif(c1)
```

但是经典辅助的 if 中是绝对不允许存在经典类型的变量的，原因是辅助类型的值是要求编译期间能够完全确定的，例如：

```
if(c) {} // Error: 编译期间无法判断 c 的值
```

### 3.1.6 2.1.6 量子程序类型 Quantum Prog Type

量子程序类型一般用于量子程序的编写与构造，在这里，我们可以简单的理解为一个操作序列。由于量子算法中也会包含经典计算，因而业界设想，最近将来出现的量子计算机是基于混合结构的，它包含两大部分：经典计算机和量子设备。经典计算机负责执行经典计算与控制；量子设备负责执行量子计算。

量子程序类型是量子编程的一个容器类，是一个量子程序的最高单位，初始化一个空的 QProg 对象有以下两种：

C++ 风格

```
qprog prog;
```

C 语言风格

```
qprog prog = CreateEmptyQProg();
```

### 3.1.7 2.1.7 函数回调类型 Callback Construct Type

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。通俗的理解就是，程序并没有调用自己定义的函数，但是在某个特定的条件下，函数却执行了（笔者的理解）。需要注意的是，如果函数回调需要传参，我们可以有两种方法避免发生错误。

方法 1: (代码为伪码描述)

将回调函数的参数作为与回调函数同等级的参数进行传递，比如：

```
circuit<int value> fun{
    value++;
}
circuit<some function, value> exe{
    some function(value);
}
exe(fun,c); //c is the parameter required for fun method
```

方法 2: (代码为伪码描述)

将回调函数的参数在调用回调函数内部创建，比如：

```
circuit<int value> fun{
    value++;
}
circuit<some function> exe{
    value={.....}; // what in the {} is custom method
    some function(value);
}
exe(fun);
```

不难看出，函数回调类型支持上述所有类型。最后笔者给出一个上述所有类型使用的程序，如下所示。有兴趣的读者可在 Qpanda 中运行并查看结果。

```

int main(void)
{
    init();
    auto qvec = qAllocMany(4);
    auto cbits = cAllocMany(4);
    auto circuit = CreateEmptyCircuit();

    circuit << H(qvec[0]) << CNOT(qvec[0], qvec[1])
        << CNOT(qvec[1], qvec[2]) << CNOT(qvec[2], qvec[3]);
    circuit.setDagger(true);
    auto prog = CreateEmptyQProg();
    prog << H(qvec[3]) << circuit << Measure(qvec[0], cbits[0]);

    auto result = runWithConfiguration(prog, cbits, 1000);
    for (auto &val : result)
    {
        std::cout << val.first << ", " << val.second << std::endl;
    }

    finalize();
    return 0;
}

```

## 3.2 2.2 字面值常量

像 42 这样的值，在经典程序中被当作字面值常量（literal constant）。称之为字面值常量是因为只能用它的值称呼它，称之为常量是因为它的值不能修改。每个字面值都有相应的类型，在 QRunes 中，支持整型、浮点型和布尔型。只有内置类型存在字面值。

### 3.3 2.2.1 整型字面值规则

在 QRunes 中定义字面值整数常量默认使用十进制，整型常量在底层都会以二进制形式表示。例如，我们将值 25 定义为整型常量：

```
25 //decimal
```

字面值整数常量的类型默认为 int 类型。它的表示范围是-32768~32767。

### 3.4 2.2.2 浮点字面值规则

在 QRunes 中定义字面值浮点常量默认使用十进制。例如，我们将值 3.14159265358979 定义为浮点常量：

```
3.14159265358979    //the default value of Pi
```

### 3.5 2.2.3 布尔字面值

单词 true 和 false 都是布尔型的字面值。

## 3.6 2.3 变量

### 3.7 2.3.1 什么是变量

QRunes 是一门静态类型语言，在编译时会作类型检查。和大多数语言一样，对象的类型限制了对象可以执行的操作。如果某种类型不支持某种操作，那么这种类型的对象也就不能执行该操作。在 QRunes 中，操作是否合法是在编译时检查的。当编写表达式时，编译器检查表达式中的对象是否按该对象的类型定义的使用方式使用。如果不是的话，那么编译器会提示错误，而不产生可执行文件。随着程序和使用的类型变得越来越复杂，我们将看到静态类型检查能帮助我们更早地发现错误。静态类型检查使得编译器必须能识别程序中的每个实体的类型。因此，QRunes 使用变量前必须先定义变量的类型。首先我们看一下什么是变量，和传统编程语言一样，变量提供了程序可以操作的有名字的存储区。QRunes 中的每一个变量都有特定的类型，该类型决定了变量的内存大小和布局、能够存储于该内存中的值的取值范围以及可应用在该变量上的操作集。我们常常把变量称为“变量”或“对象 (object)”。说到变量，难免要说到左值和右值，我们将在第 3 章详细探讨表达式，现在首先简单介绍一下 QRunes 中的两种表达式：

(1) 左值 (lvalue): 左值可以出现在赋值语句的左边或右边。

(2) 右值 (rvalue); 右值只能出现在赋值的右边，不能出现在赋值语句的左边。

变量是左值，因此可以出现在赋值语句的左边。数字字面值是右值，因此不能被赋值。给定以下变量：

```
let a=25;
let b=3.2526;
```

下面两条语句会产生编译错误：

```
a*a=b; //error: arithmetic expression is not an lvalue
0=1;  //error: literal constant is not an lvalue
```

这一部分将会在表达式章节详细介绍，此处便不再赘述。

### 3.8 2.3.2 变量名

变量名，即变量的标识符 (identifier)，可以由字母、数字组成。变量名必须以字母开头，并且严格区分大小写字母：QRunes 中的标识符都是大小写敏感的。下面例出了三个不同的标识符：

```
//three different variables
somename,someName,someName
```

在 QRunes 中并没有限制变量名的长度，但考虑到将会阅读（和 | 或）修改我们的代码的其他人，变量名不应太长。

### 3.9 2.3.3 关键字

QRunes 中保留了一组词用作改语言的关键字。关键字不能用作改语言的标识符。下面列出了所有关键字：

|         |                    |         |
|---------|--------------------|---------|
| let     | qubit              | X1      |
| include | cbit               | Y1      |
| int     | circuit            | Z1      |
| bool    | qprog              | U4      |
| if      | variationalCircuit | RX      |
| else    | hamiltonian        | RY      |
| for     | VQG_NOT            | RZ      |
| lib     | VQG_RZ             | CNOT    |
| qrunes  | VQG_RX             | CZ      |
| avar    | H                  | CR      |
| double  | X                  | CU      |
| default | NOT                | isWAP   |
| in      | T                  | measure |
| vector  | S                  | qif     |
| Pi      | Y                  | qwhile  |
| return  | Z                  | qelse   |
| lambda  | while              |         |

### 3.10 2.3.4 变量命名习惯

变量命名有很多被普遍接受的习惯，遵循这些习惯可以提高程序的可读性。

- (1) 变量名一般用小写字母。
- (2) 标识符应使用能帮助记忆的名字，也就是说，能够提示其在程序中的用法名字，如 salary.



### 3.11 2.3.5 变量的定义

变量的定义分为两个部分来说明：

#### 1. 形参变量

形参变量，只做变量声明，由传递函数的实参进行初始化，作用域为所在函数体内，当函数结束的时候，形参即被销毁。形参变量的格式：变量类型变量名当前 QRunes 支持的形参变量类型有：

|             |               |
|-------------|---------------|
| int         | hamiltionian  |
| double      | avar          |
| bool        | circuit       |
| map         | callback_type |
| qubit       |               |
| cbit        |               |
| vector_type |               |

hamiltionian 类型是哈密顿量类型数据，它是一种复合类型。

avar 是可变参数类型。

vector\_type 是数组类型的数据，具体的参数类型需要在泛型中确定。例如：vector<qubit> 表示 qubit 类型的数组。

callback\_type 是回调函数类型，由返回类型 < 参数 > 组成。例如：

```
circuit unitary(vector<qubit> q) {
    RX(q[0], -Pi);
}

//qc 为返回类型为 circuit 类型，参数类型为 vector<qubit> 的回调函数类型
circuit unitarypower(vector<qubit> q, int min, circuit<vector<qubit>> qc) {
    for (let i=0: 1: (1 << min)) {
        qc(q);
    }
}

unitarypower(q, min, unitary) //函数的调用，callback 参数类型只需传入所需调用的函数名
```

#### 2. 变量

在 QRunes 中变量的定义分为三部分来说明：

##### a. 量子类型的变量。

格式：量子类型变量名比如：

```
qubit q; => q = allocMany(1);  
vector<qubit> qvec;
```

b. 经典辅助类型的变量。

格式: `let 变量名 = 初始值`在辅助类型中的 `let` 关键字作用是定义并初始化辅助类型的变量。(占位符也是自动类型推断)。其中变量的类型由量子编译器根据初始值来推断确定变量的类型。这样做的好处:

- 1). 简化量子编程的编程操作, 并使代码简介。(凡是辅助类型的变量直接用 `let` 关键字来定义)
- 2). `let` 关键字涉及的行为只在编译期间, 而不是运行期间。

注意:

- 1). `let` 关键字定义的变量必须有初始值。

```
let a; //ERROR  
let a = 3.14; //CORRECT
```

- 2). 函数参数不可以被声明为 `let`。

```
ker(qubit q, let a){ //ERROR  
    ...  
}
```

- 3). `let` 不能与其他类型组合连用。

```
let int a = 0.09; //ERROR
```

- 4). 定义一个 `let` 关键字序列的对象的变量, 其所有初始值必须为最终能推导为同一类型。

```
let a = 0.09, b = false, c =10; //ERROR  
let a = 0.09, b = 3.14, c=100.901; //CORRECT
```

c. 经典类型的变量。

格式: 经典类型变量名比如:

```
cbit c;
```

## Chapter 4

# 第 3 章 QRunes 表达式

QRunes 提供了丰富的操作符，并定义操作数为内置类型时这些操作符的含义。掌握这些内容，我们在编程时就能有效的规避错误、提高效率。本章将重点介绍 QRunes 定义的操作符，他们使用内置类型的操作数。

表达式由一个或多个操作数 (operand) 通过操作符 (operator) 组合而成。最简单的表达式 (expression) 仅包含一个字面值常量或变量。较复杂的表达式则由操作符以及一个或多个操作数构成。

每个表达式都有一个结果 (result)。如果表达式中没有操作符，则其结果就是操作数本身 (例如，字面值常量或变量) 的值。当一个对象用在需要使用其值的地方，则计算该对象的值。例如，假设 `ival` 是一个 `int` 型对象：

```
if(ival)    //evaluate ival as a condition
// .....
```

上述语句将 `ival` 作为 `if` 语句的条件表达式。当 `ival` 为非零值时，`if` 条件成立；否则条件不成立。

对于含有操作符的表达式，它的值通过对操作数做指定操作获得。除了特殊用法外，表达式的结果是右值 (第 2 章已介绍)，可以读取该结果值，但是不允许对它进行赋值。

由我们已有的编程经验不难得知，操作符的含义就是该操作符可以执行什么操作以及操作结果的类型，它取决于操作数的类型。

除非已知道操作数的类型，否则无法确定一个特定表达式的含义。下面的表达式：

```
i + j
```

既可能是整数的加法操作或者浮点数的加法操作，也完全可能是其他的操作。如何计算该表达式的值，完全取决于 `i` 和 `j` 的数据类型。

QRunes 提供了一元操作符 (unary operator) 和二元操作符 (binary operator) 两种操作符。作用在一个操作数上的操作符称为一元操作符，如自增操作符 (`++`)；而二元操作符则作用于两个操作数上，如加法操作符 (`+`) 和减法操作符 (`-`)。除此之外，QRunes 还提供了个使用三个操作数的三元操作符 (ternary operator)，我们将在本小节最后介绍它。

操作符对其操作数的类型有要求，如果操作符应用于内置或复合类型的操作数，则由 QRunes 语言定义其类型要求。例如，自增操作符要求其操作数必须是辅助类型，对任何其他内置类型或复合类型对象进行自增操作将导致错误的产生。

对于操作数为内置或复合类型的二元操作符，通常要求它的两个操作数具有相同的数据类型。要理解由多个操作符组成的表达式，必须先理解操作符的优先级（precedence）结合性（associativity）和操作数的求值顺序（order of evaluation）。

本章节及后续的第四章来详细叙述下 QRunes 中的表达式和语句，因为当前的量子编程涉及三个部分：经典计算机模块，测控系统模块和量子芯片模块，故这种混合（量子、经典、辅助类型）程序各自分别运行在对应的硬件模块上，他们的编译和运行方式也将会不同。

## 4.1 3.1 操作符

### 4.2 3.1.1 一元操作符

| 一元操作符 | 含义      | 示例       | 支持类型 |
|-------|---------|----------|------|
| ~     | 取反操作符   | ~x       | A    |
| !     | 逻辑非操作符  | !x       | A C  |
| ++    | 一元递增操作符 | x++      | A    |
| --    | 一元递减操作符 | x--      | A    |
| .     | 方法调用操作符 | x.fun()  | A C  |
| []    | 取下标操作符  | x[value] | 数组类型 |

### 4.3 3.1.2 二元操作符

- 基本的赋值操作符是“=”。它的优先级别低于其他的操作符，所以对该操作符往往最后读取。

| 赋值操作符 | 含义            | 示例              | 支持类型 |
|-------|---------------|-----------------|------|
| =     | 将右操作数的值赋给左操作数 | x=y; 将 x 的值赋为 y | A C  |
| +=    | 加后赋值          | x+=y; 即 x=x+y   | A C  |
| -=    | 减后赋值          | x-=y; 即 x=x-y   | A C  |
| *=    | 乘后赋值          | x*=y; 即 x=x*y   | A C  |
| /=    | 除后赋值          | x/=y; 即 x=x/y   | A C  |
| %=    | 取余后赋值         | x%=y; 即 x=x%y   | A C  |
| &=    | 按位与后赋值        | x&=y; 即 x=x&y   | A C  |
| =     | 按位或后赋值        | x =y; 即 x=x y   | A C  |

- 算术操作符即算术操作符号。是完成基本的算术运算 (arithmetic operators) 符号，就是用来处理四则运算的符号。

| 算术操作符 | 含义                  | 示例       | 支持类型 |
|-------|---------------------|----------|------|
| +     | 两个操作数相加             | $x + y$  | A C  |
| -     | 第一个操作数减去第二个操作数      | $x - y$  | A C  |
| *     | 两个操作数相乘             | $x * y$  | A C  |
| /     | 第一个操作数除第二个操作数       | $x / y$  | A C  |
| %     | 第一个操作数整除第二个操作数之后的余数 | $x \% y$ | A    |
| ^     | 第一个操作数的第二个操作数幂次方    | $x ^ y$  | A    |

- 关系操作符有 6 种关系，分别为小于、小于等于、大于、等于、大于等于、不等于。关系操作符的值只能是 0 或 1。关系操作符的值为真时，结果值都为 1。关系操作符的值为假时，结果值都为 0。

| 关系操作符 | 含义                          | 示例       | 支持类型 |
|-------|-----------------------------|----------|------|
| ==    | 判断两个操作数是否相等, 相等则返回真值        | $x == y$ | A C  |
| !=    | 判断两个数是否相等, 不相等则返回真值         | $x != y$ | A C  |
| >     | 判断左操作数是否大于右操作数, 大于则返回真值     | $x > y$  | A C  |
| <     | 判断左操作数是否小于右操作数, 小于则返回真值     | $x < y$  | A C  |
| >=    | 判断左操作数是否大于等于右操作数, 大于等于则返回真值 | $x >= y$ | A C  |
| <=    | 判断左操作数是否小于等于右操作数, 小于等于则返回真值 | $x <= y$ | A C  |

- 在形式逻辑中，逻辑操作符或逻辑联结词把语句连接成更复杂的复杂语句

| 逻辑操作符 | 含义                   | 示例         | 支持类型 |
|-------|----------------------|------------|------|
| &&    | 如果两个操作数都非零, 则返回真值    | $x \&\& y$ | A C  |
|       | 如果两个操作数任意一个非零, 则返回真值 | $x    y$   | A C  |
| !     | 如果操作数为零              | $!x$       | A C  |

- 位操作是程序设计中按位或二进制数的一元和二元操作。在许多古老的微处理器上，位运算比加减运算略快，通常位运算比乘除法运算要快很多。在现代架构中，情况并非如此：位运算的运算速度通常与加法运算相同（仍然快于乘法运算）。

| 位操作符 | 含义       | 示例       | 支持类型 |
|------|----------|----------|------|
| &    | 按位与      | $x \& y$ | A    |
|      | 按位或      | $x   y$  | A    |
| ^    | 异或操作符    | $x ^ y$  | A    |
| <<   | 二进制左移操作符 | $x << y$ | A    |
| >>   | 二进制右移操作符 | $x >> y$ | A    |

## 4.4 3.1.3 三元操作符

| 三元操作符 | 含义                       | 示例                            | 类别    | 支持类型 |
|-------|--------------------------|-------------------------------|-------|------|
| ?:    | 根据计算的值结果选择 true 还是 false | <code>a &gt; b ? a : b</code> | 三元操作符 | A    |

## 4.5 3.1.4 逗号操作符

逗号操作符的作用是将几个表达式放在一起，用逗号分割，这些表达式从左向右依次计算。逗号操作符最后的结果是其最右边表达式的值。如果最右边的操作数是左值，则逗号表达式的值也是左值。此操作经常用于循环中。注：支持 A Q C

## 4.6 3.1.5 操作符详解

### 3.1.5.1 逻辑操作符与位操作符

相信有过 C 语言经验的读者知道，逻辑操作符将其操作数视为条件表达式，首先对操作数求值；若结果为 0，则条件为假（false），否则为真（true）。仅当逻辑与操作符（&&）的两个操作数都为 true，其结果才得 true。对于逻辑或操作符（||），只要两个操作数之一为 true，它的值就为 true。逻辑非操作符（!）产生与其操作数相反的条件值。如果其操作数为非 0 值，则做！操作后的结果为 false。例如：

```
expr1 &&    expr2 //logical AND
expr1 &&    expr2 //logical OR
! expr1           //logical NOT
```

仅当由 expr1 不能确定表达式的值时，才会求解 expr2（笔者注：这种求值策略被称为短路求值）。也就是说，当且仅当下列情况出现时，必须确保 expr2 是可以计算的。

- (1) 在逻辑与表达式中，expr1 的计算结果为 true。如果 expr1 的值为 false，则无论 expr2 的值为什么，逻辑与表达式的值就为 false。当 expr1 的值为 true 时，只有 expr2 的值也是 true，逻辑与表达式的值才为 true。
- (2) 在逻辑或表达式中，expr1 的计算结果为 false，则逻辑或表达式的值取决于 expr2 的值是否为 true。

辅助类型在内存中以补码的形式存储，取反操作符执行按位取反操作：二进制每一位取反，0 变 1，1 变 0。按位与操作符“&”是双目操作符，其功能是参与运算的两数各对应的二进位相与，只有对应的两个二进位均为 1 时，结果位才为 1，否则为 0。按位或操作符“|”是双目操作符。其功能是参与运算的两数各对应的二进位相或，只要对应的两个二进位有一个为 1 时，结果位就为 1。按位异或操作符“^”为双目操作符，其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为 1。左移操作符“<<”是双目操作符，左移 n 位就是乘以 2 的 n 次方，其功能把“<<”左边的运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数，高位丢弃，低位补 0。右移操作符“>>”是双目操作符。右移 n 位就是除以 2 的 n 次方，其功能是把“>>”左边的运算数的各二进位全部右移若干位，“>>”右边的数指定移

动的位数。位运算操作与 C 语言中的位运算操作并无差别，这里便不再过于赘述，只提供几个简单的小例子供读者参考。

```
00001001 & 00000101 = 00000001 // 9&5=1
00001001 | 00000101 = 00001101 // 9|5=13
00001001 ^ 00000101 = 00001100 // 9^5=12
~(1001) = 0110
x>>1;           //equivalent to x/=2
x<<1;           //equivalent to x*=2
x>>2;           //equivalent to x/=4
x<<2;           //equivalent to x*=4
x>>3;           //equivalent to x/=8
x<<3;           //equivalent to x*=8
```

需要注意的是：位运算将操作数视为二进制位的集合，为每一位提供检验和设置的功能，它只适用于辅助类型，对其它数据类型不适用；逻辑运算表达式的结果只能是 1 或 0，而位运算的结果可以取 0 或 1 以外的值。移位操作的右操作数不可以是负数，而且必须是严格小于左操作数位数。否则，操作的效果未定义。

### 3.1.5.2 关系操作符

关系操作符使用辅助类型或经典类型的操作数，并返回布尔型的值。关系操作符的值为 true 时，结果值为 1；关系操作符的值为 false 时，结果值为 0。点操作符 (.) 一般运用于方法调用，取下标运算 ([]) 用于获取数组类型中某个特定索引对象的具体信息。

### 3.1.5.3 赋值操作符

赋值操作符的操作与 C 语言相同，在这里我们屏蔽其具体操作，而详细阐述其合法性要求。

赋值操作符的左操作数必须是非常量的左值，下面的赋值语句是不合法的：

```
let i=1,j=2,ival=3;
let ci=i;    // ok:initialization not assignment
2048=ival;   // error:literal are rvalues
i + j = ival; //error:arithmetic expressions are rvalues
ci = ival;   //error:can't write to ci
```

在数组类型中，数组名是不可修改的左值：因此数组不可以做赋值操作的目标。而下标操作符也返回左值，因此这种操作位于非静态数组时，其结果可以作为赋值结果的左操作数：

```
vector<int> ives;
ives[0] = 1024; // ok: subscript is an value
```

我们应该注意到，赋值表达式的值是其左操作数的值，其结果的类型为左操作数的类型。通常情况下，赋值操作将其右操作数的值赋给左操作数，然而当左右操作符类型不同时，该操作将无法实现。因此我们在编程时，可以事先人工检查类型是否一致。

### 3.1.5.4 算术操作符

对于算术操作符来说，我们首先考虑此类操作符的优先级，一元操作符的优先级最高。其次是乘、除操作，接着是二元的加、减法操作。高优先级的操作符要比低优先级的结合的更紧密。上述表格中的所有算符都是左结合，这就意味着当操作符的优先级相同时，这些操作符从左向右依次与操作数结合。关于算符的优先级关系，我们将在下面仔细阐述。以下是一个简单的小例子：

```
3 + 10 * 20 / 2;
```

考虑到优先级和结合性，可知该表达式先做乘法（\*）操作，其操作数是 10 和 20，然后以该操作的结果和 2 为操作数做除法（/）操作，其结果最后与操作数 3 做加法（+）操作。

关于算符表达式，考虑到其中的数学特性，我们应避免诸如除零操作等违法操作，对于计算机特性而言，比如溢出和截断，在这里我们将不予考虑。

### 3.1.5.5 自增和自减操作符

自增（++）和自减（--）操作符为对象加 1 或减 1 操作提供了方便简短的实现方式。和 C 一样，在 QRunes 中它们也有前置和后置两种使用形式。这里在简单重复一下前置和后置形式的微小差别。前自增操作使其操作数加 1，操作结果是修改后的值。同理，前自减操作执行操作数减 1 操作。这两种操作符的后置形式同样对其操作数加 1（或减 1），但操作后产生操作数原来的、未修改的值作为表达式的结果，此类操作只支持辅助类型对象：

```
let i = 0;
let j = ++i; // j=1, i=1. prefix yields incremented value
let k = --i; // k=1, i=2. prefix yields unincremented value
```

我们不难发现，前自增操作要做的工作更少，只需要加 1 后返回加 1 的结果即可。而后自增操作则需要保存操作数原来的值，以便返回未加 1 之前的值作为操作的结果。因此，有经验的程序员在处理复杂类型对象时，都会优先使用前置操作。对于在单个表达式中组合使用此类操作，和 C++ 的操作一致。

## 4.7 3.2 表达式

在 QRunes 中，表达式由操作符和操作数组成，主要的作用是：

- 计算辅助类型操作数的值
- 指定函数

操作数可以是常量或者一个数据对象。比如：

- 常量：3.14, 1
- 数据对象：标识符, 表达式本身

### 4.8 3.2.1 主表达式

它是构造其他表达式的基本块。



语法构成：

主表达式：标识符 | 常量 | 括号表达式  
`primary_expression: identifier | constant | parenthesis_expression`  
 例如：`qubit_s1, 3.1415, (c1 + c2)`  
 注：支持量子类型，经典类型，辅助类型

## 4.9 3.2.2 括号表达式

语法构成：

`parenthesis_expression: ( expression )`  
 它表示在不更改括号封闭里面的表达式类型或值的情况下构造表达式的分组方式。  
 例如： `( 2 + 3 ) / 5` 与 `2 + 3 / 5`  
 注：支持量子类型，经典类型，辅助类型

## 4.10 3.2.3 复合表达式

含有两个或更多操作符的表达式称为复合表达式（compound expression）。在复合表达式中，操作数与操作符的结合方式决定了整个表达式的值。表达式的结果会因为操作符与操作数的分组结合方式的不同而不同。操作数的分组方式取决于操作符的优先级和结合性。也就是说，优先级和结合性决定了表达式的哪个部分用作哪个操作符的操作数。如果程序员不想考虑这些规则，可以在复合表达式中使用圆括号强制实现某个特定的分组。一般情况下，优先级规定的是操作数的结合方式，但并没有说明操作数的计算顺序。

### 3.2.3.1 优先级和结合性

表达式的值取决于其子表达式如何分组；优先级规定了具有相同优先级的操作符如何分组。下表给出了 QRunes 中操作符的优先级和结合性。

| 操作符                                   | 结合性  |
|---------------------------------------|------|
| <code>[ ] . ( )</code> (方法调用)         | 从左向右 |
| <code>! ~ ++ -- +(一元运算)-(一元运算)</code> | 从右向左 |
| <code>* / %</code>                    | 从左向右 |
| <code>+-</code>                       | 从左向右 |
| <code>&lt;&lt; &gt;&gt;</code>        | 从左向右 |
| <code>&lt; &lt;= &gt; &gt;=</code>    | 从左向右 |
| <code>== !=</code>                    | 从左向右 |
| <code>&amp;&amp;   </code>            | 从左向右 |
| <code>?:</code>                       | 从右向左 |
| <code>= += -= *= /= %=</code>         | 从右向左 |

当然，我们可以使用圆括号来推翻操作符优先级的限制，使用圆括号的表达式将用圆括号括起来的子表达式视为独立单元先进行计算，其他的部分则以普通优先级规则处理。下面对于操作符的优先级和结合性给出几个例子：

```
ival1 = ival2 = ival3 = ival4    //right associative
(ival1 = (ival2 =(ival3 =ival4))) //equivalent,parenthesized version
ival1 * ival2 / ival3 * ival4    //left associative
(((ival1 * ival2) / ival3 * ival4) //equivalent,parenthesized version
```

### 3.2.3.2 注意事项

大多数操作符没有规定其操作数的求值顺序：由编译器自由选择先计算左操作数还是右操作数。通常操作数的求值顺序不会影响表达式的结果。但是，如果操作符的两个操作数都与同一个对象有关，而且其中一个操作数改变了该对象的值，则程序将会因此而产生严重的错误，并且此类错误很难被发现。

## 4.11 3.2.4 常量表达式

常量表达式是在编译时计算而不是在运行时计算。

注：支持 A

## 4.12 3.2.5 lambda 表达式

匿名函数 lambda：是指一类无需定义标识符（函数名）的函数或子程序。

lambda 函数可以接收任意多个参数（包括可选参数）并且返回单个表达式的值。

lambda 匿名函数的格式：冒号前是参数，可以有多个，用逗号隔开，冒号右边的为表达式或是语法块。其实 lambda 返回值是一个函数的地址，也就是函数对象。

示例：

```
circuit<vector<qubit>,qubit> generate_two_qubit_oracle(vector<bool> oracle_function){
    return lambda (vector<qubit> qlist,qubit qubit2):{
        if (oracle_function[0] == false &&
            oracle_function[1] == true){
            // f(x) = x;
            CNOT(qlist[0], qubit2);
```

(continues on next page)

(continued from previous page)

```

    }else if (oracle_function[0] == true &&
        oracle_function[1] == false){
        // f(x) = x + 1;
        CNOT(qlist[0], qubit2);
        X(qubit2);
    }else if (oracle_function[0] == true &&
        oracle_function[1] == true){
        // f(x) = 1
        X(qubit2);
    }else{
        // f(x) = 0, do nothing
    }
};
}

Deutsch_Jozsa_algorithm(vector<qubit> qlist,qubit qubit2,vector<cbit> clist,circuit
↪<vector<qubit>,qubit> oracle){
    X(qubit2);
    apply_QGate(qlist, H);
    H(qubit2);
    oracle(qlist,qubit2);
    apply_QGate(qlist, H);
    measure_all(qlist,clist);
}

```

注意：lambda 表达式包含的语法块或表达式不能超过一个



## Chapter 5

# 第 4 章语句

- QRunes 中语句近似与人类的自然语言，既有完成单一任务的简单语句也有作为一个集合的一组语句组成的复合语句。同时 QRunes 既能支持辅助类型的条件分支和循环控制结构，也支持经典类型的 QIF 和 QWHILE 的量子分支和量子循环控制结构。
- QRunes 中的语句大部分都是以分号；结尾。

### 5.1 4.1 简单语句

#### 5.1.1 4.1.1 表达式语句

表达式语句的类型取决于表达式类型，表达式类型参见 Chapter 3.2。

举例如下：

```
c2 = c1 + 1; //Assign statement
```

#### 5.1.2 4.1.2 声明语句

- QRunes 中声明语句主要分为两种：函数的声明和变量的定义与。
- 具体可以参见 Chapter 2.3 和 Chapter 5.2，其中变量的定义支持 A Q C 类型的变量定义。

举例如下：

```
qubit q; // declaration a variable with qubit type named q
let i = 3.14; // declaration a variable with assist-classical type which named i and
↳ initialized 3.14
```

## 5.2 4.2 复合语句

- QRunes 中复合语句通常按块的概念，表现形式为使用一对花括号 {} 括起来的语句序列。
- 在复合语句中的一组语句不仅仅是一堆简单语句的组合，同时根据程序的逻辑要求，一个程序逻辑也被称为一条语句块，比如 if,for,qif,qwhile。

例如：

```
if (oracle_function[0] == False && oracle_function[1] == True) {
    // f(x) = x;
    CNOT(q1, q2);
} else if (oracle_function[0] == True && oracle_function[1] == False) {
    // f(x) = x + 1;
    X(q2);
    CNOT(q1, q2);
    X(q2);
}
```

注意：

1. 与其他语句不同的是，复合语句不是以分号；结尾。
2. 只能用在某个函数体中书写。

## 5.3 4.3 函数调用语句

### 5.3.1 4.3.1 量子逻辑门操作函数调用语句

- 在 QRunes 中所有对于量子比特的操作，我们称为逻辑门函数或者 XX 门，比如我们常说的 X 门，Y 门，CNOT 门，他们都是类似于 QRunes 的库文件中实现过其函数实现，预先定义好的，用户可以直接通过调用的形式实现逻辑门的操作。
- 当前 QRunes 支持 18 中量子逻辑门函数的操作，其函数声明分别如下：

```
H(qubit);
NOT(qubit);
T(qubit);
S(qubit);
Y(qubit);
Z(qubit);
X1(qubit);
Y1(qubit);
```

(continues on next page)

(continued from previous page)

```

Z1(qubit);
U4(qubit,alpha,beta,gamma,delta);
RX(qubit,alpha);
RY(qubit,alpha);
RZ(qubit,alpha);
CNOT(qubit,qubit);
CZ(qubit,qubit);
CU(qubit,qubit,alpha,beta,gamma,delta);
ISwap(qubit,qubit,alpha);
CR(qubit,qubit,alpha);

```

### 5.3.2 4.3.2 可变量子逻辑门函数调用语句

可变量子逻辑门是构成可变量子线路 VQC 的基本单位, 可变量子逻辑门函数内部维护着一组变量参数以及一组常量参数。当前 QRunes 支持 6 种可变量子逻辑门函数调用:

```

VQG_H(qubit);
VQG_RX(qubit,alpha);
VQG_RY(qubit,alpha);
VQG_RZ(qubit,alpha);
VQG_CNOT(qubit,qubit);
VQG_CZ(qubit,qubit);

```

### 5.3.3 4.3.3 经典返回值类型函数调用语句

定义一个函数, 但是该函数并不会自动的执行。定义了函数仅仅是赋予函数以名称并明确函数被调用时该做些什么。调用函数才会以给定的参数真正执行这些动作, 比如如下函数:

```

Reset_Qubit_Circuit(qubit q, cbit c, bool setVal) {
    Measure(q, c);
    if (setVal == False) {
        qif (c) {
            X(q);
        }
    } else {
        qif (c) {
        } qelse {
            X(q);
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
}
Reset_Qubit(qubit q, cbit c, bool setVal) {
// quantum logic gate function call, and can reference to its function definition
    Reset_Qubit_Circuit(q, c, setVal);
}

```

其中的 `Reset_Qubit_Circuit` 函数在 `Reset_Qubit` 中的调用，该表示方法就是函数调用。

注意：

1. 函数调用语句必须严格按照函数调用的格式进行书写：

```
function_name(args....);
```

2. 回调函数中的参数必须严格匹配原函数定义中的参数的类型、个数。
3. 函数调用语句只能在调用函数体内书写。

## 5.4 4.4 辅助类型控制语句

### 5.4.1 4.4.1 选择语句

QRunes 中的选择语句主要是 `if-else` 格式的语句，其计算流程为根据 `if` 中表达式的是否有条件地执行分支语句，其中 `else` 分支可以是可选项。

语法结构如下：

```

if(condition)
    statement;
else
    statement;

```

举例如下：

```

if (fx) {
    X(q[0]);
}else{
    H(q[0]);
    X(q[1]);
}

```



其中 if 中的 condition 必须是一个返回值为 bool 类型的表达式或者可以转换为 bool 类型的表达式，此外 statement 部分可以用花括号括起来的复合语句。

### 5.4.2 4.4.2 while 循环语句

- 在循环刚开始时，会计算一次“布尔表达式”的值，若条件为真，执行循环体。而对于后来每一次额外的循环，都会在开始前重新计算一次。while 语句会反复地进行条件判断，直到条件不成立，while 循环结束。
- 语句中应有使循环趋向于结束的语句，否则会出现无限循环——“死”循环。

语法结构如下：

```
while (condition) {
    statement;
}
```

举例如下：

```
while(data >= 1){
    if(data % 2 == 1){
        X(qlist[i]);
    }
    data = data >> 1;
    i = i + 1;
}
```

其中 while 中的 condition 必须是一个返回值为 bool 类型的表达式或者可以转换为 bool 类型的表达式，此外 statement 部分可以用花括号括起来的复合语句。

### 5.4.3 4.4.3 循环语句

QRunes 中的循环语句主要是 for 循环语句，其语法格式如下：

```
for(initializer:condition:expression)
    statement
```

其中 initializer、condition 和 expression 都是以冒号结束，initializer 用于循环结构的变量初始化；condition(循环条件)则是用来控制循环的，当判断条件为 true 的时候则执行 statement；expression 用来修改 initializer 的值。特殊情况如下，当循环结构第一次在求解 condition 的时候就返回 false，则该循环体将始终不会执行。通常，循环体中的 statement 可以是单个语句也可以是复合语句。

举例如下：

```
for(let i=0: 1: qlist.size()){  
    VQG_RX(qlist[i],2.0*beta);  
}
```

展示的程序用将以 qubit 为类型的集合 qlist 中的每个 qubit 进行可变量子线路构造的操作。

## 5.5 4.5 量子类型控制语句

### 5.5.1 4.5.1 QIF 语句

QIF 的结构如下：

```
qif(condition)  
    statement  
qelse  
    statment
```

与 4.4.1 中的 if 相比较，二者的差别在 condition 和 statement 中，QIF 语句中的 condition 必须是经典类型且返回值为 bool 类型的表达式，statement 只能为返回值为经典类型的语句、量子逻辑门操作函数调用语句、返回值为量子类型（QProg, QCircuit）的函数调用、量子比特测量语句和 QIF/QWHILE 语句。

举例如下：

```
qif(!c1){  
    Measure(q[2],c[2]);  
}  
qelse{  
    Measure(q[1],c[1]);  
}
```

### 5.5.2 4.5.2 QWHILE 语句

QWHIE 的结构如下：

```
qwhile(condition)  
    statement
```

与 4.4.2 中的 for 相比较，二者的差别在 condition 和 statement 中，QWHILE 语句中的 condition 必须是经典类型且返回值为 bool 类型的表达式，statement 只能为返回值为经典类型的语句、量子逻辑门操作函数调用语句、返回值为量子类型（QProg, QCircuit）的函数调用、量子比特测量语句和 QIF/QWHILE 语句。

举例如下：

```

qwhile(c[0] < 3){ //c is declared by type cbit
    H(qvec[c[0]]);
    let i = 1; //the value of declaration statement is assist-classical.EEROR!
    c[0] = c[i] + 1; //ERROR?
    c[0] = c[0] + 1;
}

```

## 5.6 4.6 量子比特测量语句

量子测量是指通过量子计算机的测控系统对量子系统进行干扰来获取需要的信息，测量比特使用的是蒙特卡罗方法的测量。QRunes 中的量子比特测量语句的结构如下：

```
measure(qubitType,cbitType);
```

举例如下：

```

H(q);
measure(q,c);

```

## 5.7 4.7 return 语句

- return 语句作为一个无条件的分支，无需判断条件即可发生。
- return 语句：是指结束该方法，继续执行方法后的语句。

```
return statement;
```

举例如下：

```
return 0;
```



# Chapter 6

## 第 5 章函数

在 QRunes 中，函数可以被看作是由开发人员自己定义的为了完成某个功能任务的一组语句的集合，每个 QRunes 程序都会至多有一个入口函数，在 QRunes 中入口函数即 main 函数都是在第三方经典语言中来进行实现。

### 6.1 5.1 函数的声明

函数声明和变量的定义和声明一样，必须是声明之后才可以使用。函数的声明很函数的定义可以分离开来，同时一个函数只能定义一次，但是可以声明多次。

函数声明的结构如下：

```
return_type? function_name(args);
```

举例如下：

```
circuit createCircuit(qubit qu);
```

返回值，函数名和函数形参列表称为函数原型。函数的声明为定义函数的使用者和调用函数的使用者之间提供了一种天然的接口。

### 6.2 5.2 函数的定义

QRunes 中，函数是由返回值，函数名，函数参数和一组语句组成。其中函数名在同一个.qrunes 文件中必须唯一；函数参数即函数的形式参数，它们由一对圆括号包围并在其中进行声明，形参之间的以逗号进行分割；一组语句即函数的函数体是函数的执行部分。每一个函数都有一个相关联的返回类型。

函数定义的格式如下：

```
return_type? function_name(args){
    // function_body
}
```

举例如下：

```
Two_Qubit_DJ_Algorithm_Circuit(qubit q1, qubit q2, cbit c, vector<bool> oracle_function)
↪{
    H(q1);
    measure(q1, c);
}
```

### 6.3 5.3 函数的参数

- QRunes 中函数不能省略或者为空，函数的形参表由一系列的由逗号分隔符分离的参数类型和参数名组成，如果两个形参的类型相同，则其类型必须重复声明。
- 在 QRunes 中所有的函数参数都必须命名之后才可以使用。
- 形参和实参：类似于局部变量，函数的形参为函数提供了已命名的局部存储空间。他们之间的差别在于形参是在函数的形参表中定义的，并由调用函数时传递给函数的实参初始化。

### 6.4 5.4 函数的返回值

QRunes 中函数的返回类型可以是内置类型、复合类型也可以是 void 类型。其中，

- 内置类型

```
qprog
circuit
variationalCircuit
int
double
bool
cbit
```

下面给出一个返回值为 qprog 的函数供读者参考：

```
BV_QProg(vector<qubit> qVec, vector<cbit> cVec, vector<bool> a, circuit<vector<qubit>,
↪qubit> oracle){
```

(continues on next page)

(continued from previous page)

```

if(qVec.length() != (a.length()+1)){

}

let cd = qVec.length();

X(qVec[cd-1]);
apply_QGate(qVec, H);
oracle(qVec, qVec[cd - 1]);

qVec.pop();

apply_QGate(qVec, H);
measure_all(qVec, cVec);

}

```

- 复合类型

复合类型即由 `vector` 关键字构造的类型集合，其中的 `vector` 集合中的类型为经典类型。

比如：

```
vector<cbit>
```

下面给出一个例子工读者参考：

```

circuit<vector<qubit>,qubit> generate_3_qubit_oracle(int target){
    return lambda (vector<qubit> qvec,qubit qu):{
        if(target == 0){
            X(qvec[0]);
            X(qvec[1]);
            Toffoli(qvec[0], qvec[1], qu);
            X(qvec[0]);
            X(qvec[1]);
        }
        if(target == 1){
            X(qvec[0]);
            Toffoli(qvec[0], qvec[1], qu);
            X(qvec[0]);
        }
        if(target == 2){

```

(continues on next page)

(continued from previous page)

```

        X(qvec[1]);
        Toffoli(qvec[0], qvec[1], qu);
        X(qvec[1]);
    }
    if(target == 3){
        Toffoli(qvec[0], qvec[1], qu);
    }
};

```

```

}

```

- **void 类型**

不带返回值的 return 语句只能用于返回类型为 void 的函数。在返回类型为 void 的函数中，return 返回语句不是必需的。下面给出一个 C++ 的例子：

```

void swap(int& a, int&b)
{
    if(a == b)
    { return;
    } int temp; temp = a; a = b; b = temp;
}

```

- **函数回调类型**

callback\_type 是回调函数类型，由返回类型 < 参数 > 组成。

比如：

```

circuit<vector<qubit> >

```

注：最右边的两个尖括号 (> >) 在中间应该要加一个空格。如果空格不存在，则在编译时会存在错误。

根据函数的返回值可以将 QRunes 中的函数分为两个部分：量子函数和经典函数。其中的返回值为经典类型、经典类型构造的集合类型和 void 类型为经典函数，其余为量子函数。

函数的定义举例如下：

```

//quantum function
qu_function(vector<qubit> qvec, vector<cbit> cvec){
    for(let i = 0: 1: len(qvec)){
        H(qvec[i]);
        Measure(qvec[i], cvec[i]);
    }
}

```

(continues on next page)



(continued from previous page)

```

vector<cbit> cc = getCbitNotEqualZero(cvec);
for(let c in cc){
    c = c + 1;
}
}

```

```

//classical function
vector<cbit> getCbitNotEqualZero(vector<cbit> cvec){
    vector<cbit> c2;
    for(let c in cvec){
        if(c == 1){
            c = c + 1;
            c2.insert(c);
        }
    }
    return c2;
}

```

```

//return value is null
void rotateOperation(vector<qubit> qlist){
    for(let qu in qlist){
        H(qlist[i]);
    }
}

```

## 6.5 5.5 函数调用

函数调用的结构：

```
function_name(args...);
```

其中的实参可以是常量，变量，多个实参之间用逗号进行分割。

函数调用的方式：

- 函数调用作为表达式中的一项，常用于赋值表达式，也可称为函数调用表达式

举例：

```
c = getCbit(cbit c);
```

- 函数作为单独的语句，及函数调用语句

举例：

```
ker(qlist,clist);
```

- 函数也可以作为另一个函数的实参

## Chapter 7

# 第 6 章常用量子算法介绍及实现

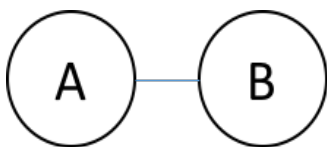
## 7.1 6.1 QAOA 算法

### 7.1.1 6.1.1 QAOA 算法介绍

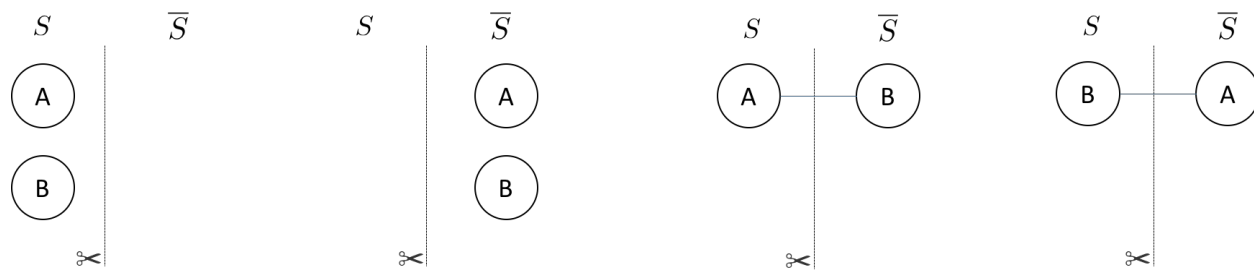
量子近似优化算法 (QAOA), 是由 Farhi, Goldstone 和 Gutmann 开发的一个多项式时间算法, 用于寻找“最优化问题的一种‘好’的解决方案”。对于给定的 NP-Hard 问题, 近似算法是一种多项式时间算法, QAOA 算法以期望的一些质量保证来解决每个问题实例。品质因数是多项式时间解的质量与真实解的质量之间的比率。

#### NP-Hard Problem

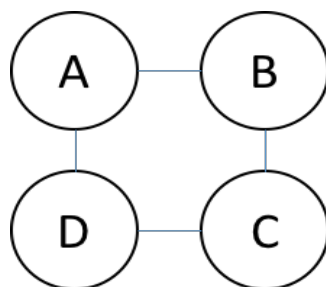
最大切割问题 (MAX-CUT) 是原始量子近似优化算法论文中描述的第一个应用。此问题类似于图形着色。给定节点和边的图形, 将每个节点着色为黑色或白色, 然后对有不同颜色节点的边给定一个分值。目的是找到得分最高的着色点。更正式地说, 问题是将图的节点划分为两组, 使得连接相对组中的节点的边的数量最大化。例如, 杠铃图



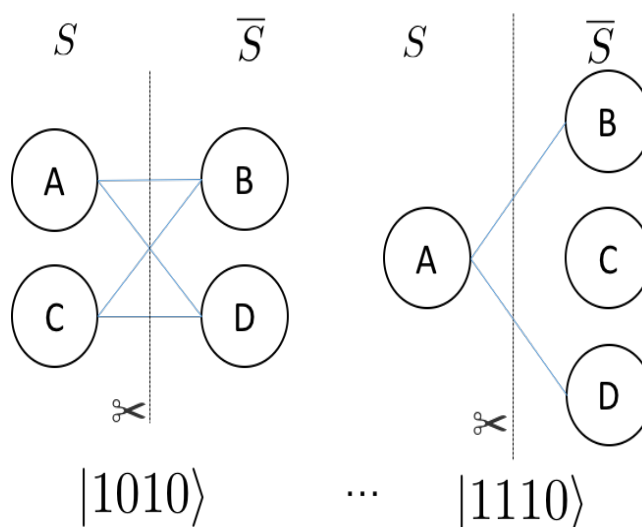
有 4 种方法将节点分为两组:



我们仅对连接不同集合中的节点时才会绘制边。带有剪刀符号的线条表示我们要计算的切割边。对于杠铃图，有两个相等的权重分区对应于最大切割（如上图，右侧两个分区），将杠铃切成两半。我们可以将集合  $S$  或  $\bar{S}$  中的节点表示为 0 或 1，组成一个长度为  $N$  的比特串。上图的四个分区可以表示为  $\{00, 11, 01, 10\}$  其中最左边的比特对应节点 A，最右边的比特对应节点 B。用比特串来表示使得表示图的特定分区变得很容易。每个比特串具有相关联的切割权重。对任意一个图中，节点分割所使用的比特串长度总是  $N$ 。可分割的情况总数是  $2^N$ 。例如，方形环见下图：



有 16 个可能的分区 ( $2^4$ )。以下是两种可能的节点分区方式：



与每个分区相关联的比特串如上图所示，其中最右边的比特对应节点 A，最左边的比特对应节点 D。

## 解决方案

### 经典解决算法

为了在经典计算机上找到最佳剪切，最直接的方法是枚举图的所有分区并检查与分区关联的切割的权重。

面对寻找最佳切割（或一组最佳切割）的指数成本，我们可以设计出具有特定质量的多项式算法。例如，著名的多项式时间算法是随机分区方法，通过遍历图表的节点并抛掷硬币，如果硬币是正面，则节点在集合  $S$  中，否则节点在集合  $\bar{S}$  中。随机分配算法的质量至少是最大切割的 50%。对于硬币抛掷过程来说，边被切割中的概率是 50%。因此，随机分配产生的切割的期望值可以写成如下：

$$\sum W_e \cdot Pr(e \in cut) = \frac{1}{2} \sum W_e$$

由于所有边的总和必然是最大切割的上限，因此随机化方法产生的切割预期值至少是中最佳切割的 0.5 倍。其它多项式方法包括半定规划可以使期望的切割值至少为最大切割的 0.87856 倍。

### 量子近似优化算法 (QAOA)

我们可以将对应于图上的最大切割的比特串（或比特串组），视为是使用哈密顿量编码的代价函数的基态。这个哈密顿量的形式，可以通过构造经典函数返回值来决定，如果被切割边所对应的两个节点跨越不同集合，则返回 1（或边的权重），否则返回 0。

$$C_{ij} = 1/2(1 - z_i z_j)$$

如果节点  $i$  和节点  $j$  属于  $S$ ，则  $z_i$  或  $z_j$  的值为 +1，否则为 -1。总代价是图中所有  $(i, j)$  节点对形成的边集权重的总和。这表明对于 MAX-CUT，哈密顿量编码的问题是

$$\sum_{ij} \frac{1}{2}(I - z_i z_j)$$

其中和是对图中  $(i, j)$  节点对组成的边集权重的遍历。量子近似优化算法 (QAOA) 基于这样的事实：我们可以准备一些近似于该哈密顿量的基态的东西并对该状态进行测量，对  $N$  位量子状态执行测量，并以高概率返回对应于最大切割的比特串。为了使描述更具体，让我们回到杠铃图。该图需要两个量子位以表示节点。就会有如下的哈密顿量的形式

$$\hat{H} = \frac{1}{2}(I - \begin{matrix} 1 & 0 \\ z & z \end{matrix}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

基序对应于以二进制格式增加整数值（最左边的位是最重要的）。这对应于上面的  $\hat{H}$  操作符的基底是：

$$(|00\rangle, |01\rangle, |10\rangle, |11\rangle)$$

这里哈密顿量是带有整数特征值的对角线，显然这里每个比特串都是哈密顿量的特征态因为  $\hat{H}$  是对角线的。

QAOA 通过从参考状态演变来识别 MAXCUT 哈密顿量的基态，该参考状态是哈密顿量的基态，它耦合总共  $2^N$  个状态来构成本哈密顿量的基底——成本函数的对角基底。对于 MAX-CUT，这是以  $Z$  为计算基底。

参考哈密顿量的基态与 MAXCUT 哈密顿量的基态之间的演化可以通过两个算子之间的插值产生

$$\hat{H} = \hat{H}_{ref} + (1 - \tau)\hat{H}_{MAXCUT}$$

其中  $\tau$  在 1 和 0 之间变化，如果参考的哈密顿量基态已经准备好并且  $\tau=1$ ，这个状态是  $\hat{H}$  的静止状态。当  $\hat{H}$  变换到 MAXCUT 哈密顿量，基态相对于  $\hat{H}(1)$  将演变成为不再静止。这可以被看作是 QAOA 演变的连续版本。

算法的近似部分来自于使用多少个  $T$  值来逼近连续演化。我们将这个切片数称作  $\alpha$ 。原始论文证明，对于  $\alpha = 1$  时，最优线路产生的状态分布，具有哈密顿期望值为 3-正则图的真实最大切割值 0.6924。此外，通过增加近似演化的切片数量，可以改善真实最大切割与来自 QAOA 的期望值之间的比值。

### 实现细节

对于 MAXCUT，参考哈密顿量是  $x$  运算符在每个量子位上的和。

$$\hat{H}_{ref} = \sum_{i=0}^{N-1} \sigma_i^X$$

该哈密顿量具有基态，该基态是  $x$  算子 ( $|+\rangle$ ) 的最低特征向量的张量积。

$$|_{ref} = |+\rangle_{N-1} |+\rangle_{N-2} \dots |+\rangle_0$$

通过在每个量子位上执行 Hadamard 门很容易生成参考状态。假设系统的初始状态全为零。生成此状态的 QRunes 代码是

```
H 0
H 1
...
H N-1
```

QPanda::QAOA 要求用户输入参考哈密顿量和 MAXCUT 哈密顿量之间演化的近似步长。然后，该算法使用最大化成本函数的 quantum-variational-eigensolver 方法来变分地确定旋转的参数（表示为  $\beta$  和  $\gamma$ ）。

例如，如果选择 ( $\alpha = 2$ )，则生成近似连续演化的两个单一算子。

$$U = U(\hat{H}_{\alpha 1})U(\hat{H}_{\alpha 0})$$

每个  $U(\hat{H}_{\alpha i})$  由一阶 Trotter-Suzuki 分解近似，Trotter 步数等于 1

$$U(\hat{H}_{si}) = U(\hat{H}_{ref,i})U(\hat{H}_{MAXCUT,i})$$

其中

$$U(\hat{H}_{ref,i}) = e^{-i_i \hat{H}_{ref}}$$

并且

$$U(\hat{H}_{MAXCUT,i}) = e^{-i_i \hat{H}_{MAXCUT}}$$

$U(\hat{H}_{ref,i})$  和  $U(\hat{H}_{MAXCUT,i})$  可以表示为一个短的量子线路。

对于  $U(\hat{H}_{ref,i})$  项（或混合项），总和中的所有运算符都可以通信，因此可以分成指数  $x$  运算符的乘积。

$$e^{-i_i \hat{H}_{ref}} = \prod_{n=0}^1 e^{-i_i \sigma_n^x}$$

```
H 0
RZ(beta_i) 0
H 0
H 1
RZ(beta_i) 1
H 1
```

当然，如果 RZ 在量子处理器的自然门集中，则该 QRunes 被编译成一组 RZ 旋转。QRunes 代码风格的成本函数。

$$e^{-i\frac{\gamma}{2}(I - \frac{Z}{1} \frac{Z}{0})}$$

看起来像这样：

```
X 0
U1(gamma{i}/2) 0
X 0
U1(gamma{i}/2) 0
CNOT 0 1
RZ(gamma{i}) 1
CNOT 0 1
```

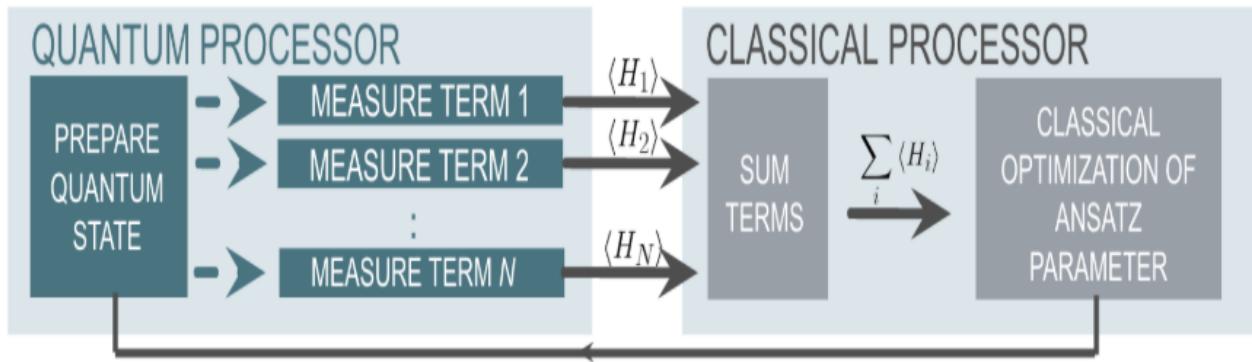
执行 QRuns 代码将会产生  $|+_1\rangle + |_0\rangle$  状态，并用选定的  $\beta$  和  $\gamma$  角度进行演化

$$|_+ = e^{-i_1 \hat{H}_{ref}} e^{-i_1 \hat{H}_{MAXCUT}} e^{-i_0 \hat{H}_{ref}} e^{-i_0 \hat{H}_{MAXCUT}} |_{+_{N-1}, \dots, 0}$$

为了识别最大化目标函数的  $\beta$  和  $\gamma$  角的集合

$$, |\hat{H}_{MAXCUT}|,$$

QPanda::QAOA 利用经典量子混合方法，称为 quantum-variational-eigensolver。量子处理器通常通过多项式操作来准备状态，然后使用该操作来评估成本。评估成本 ( $|\hat{H}_{MAXCUT}|$ ) 需要进行许多准备和测量以生成足够的样本来准确地构建分布。然后，经典计算机再生成一组新的参数 ( $\beta, \gamma$ )，以最大化成本函数。



通过允许  $\beta$  和  $\gamma$  角度的自由变化，QAOA 找到固定步数的最佳路径。一旦通过经典优化循环确定了最佳角度，就可以通过  $\beta$ 、 $\gamma$  和采样的状态的许多准备来读出分布。

### 7.1.2 6.1.2 QAOA 算法的实现

下面给出 QRunes 实现 QAOA 算法的代码示例：

Python

```

@settings:
    language = Python;
    autoimport = True;
    compile_only = False;

@qcodes:
//Solving the problem of Maximum Cutting
variationalCircuit oneCircuit(vector<qubit> qlist, hamiltonian hp, avar beta, avar gamma)
↪{
    for(let i = 0 : 1: hp.size()){
        vector<qubit> tmp_vec;
        let hamiltonItem = hp[i];
        let dict_p = hamiltonItem.getFirst();
        for(map m in dict_p) {
            tmp_vec.add(qlist[m.first()]);
        }

        let coef = tmp_vec.getSecond();
        VQG_CNOT(tmp_vec[0], tmp_vec[1]);
        VQG_RZ(tmp_vec[1], 2*gamma);
        VQG_CNOT(tmp_vec[0], tmp_vec[1]);
    }

    for(let i=0: 1: qlist.size()){
        VQG_RX(qlist[i], 2.0*beta);
    }
}

@script:
import numpy as np

#Convert the data format to be processed
def trans(friendShip):
    pro = {}
    for i in range(len(friendShip)):
        for j in range (len(friendShip[i])):
            if i != j:
                s = "Z" + str(i) + " " + "Z" + str(j)
                pro[s] = friendShip[i][j]
    return pro

```

(continues on next page)



(continued from previous page)

```

if __name__=="__main__":
    firendShip = [[0, 0.8, 0.2, -0.2],[0.8, 0, 0, 0.7],[0.2, 0, 0, -0.3],[-0.2, 0.7, -0.3,
↪ 0]]
    print("what we r need to handle:")
    print(firendShip)
    problem = trans(firendShip)

    #Bulid pauli operator base on the data of problem
    Hp = PauliOperator(problem)
    qubit_num = Hp.getMaxIndex()

    machine = init_quantum_machine(QMachineType.CPU_SINGLE_THREAD)
    qlist = machine.qAlloc_many(qubit_num)
    step = 4
    beta = var(np.ones((step,1), dtype = 'float64'), True)
    gamma = var(np.ones((step,1), dtype = 'float64'), True)

    #Create a variable quantum circuit
    vqc = VariationalQuantumCircuit()

    #Insert Hadamard gates to each qubit as initial condition
    for i in qlist:
        vqc.insert(VariationalQuantumGate_H(i))

    #Insert quantum circuits corresponding to each step according to the step size
    for i in range(step):
        vqc.insert(oneCircuit(qlist, Hp.toHamiltonian(1), beta[i], gamma[i]))

    #Calculate loss variables
    loss = qop(vqc, Hp, machine, qlist)
    #Use momentum-based optimizer and get result variables
    optimizer = MomentumOptimizer.minimize(loss, 0.02, 0.9)
    leaves = optimizer.get_variables()

    for i in range(100):
        loss_value = optimizer.get_loss()
        print("i: ", i, " loss:", loss_value )
        optimizer.run(leaves, 0)

```

(continues on next page)

(continued from previous page)

```

prog = QProg()
qcir = vqc.feed()
prog.insert(qcir)
#Run quantum programs
directly_run(prog)

result = quick_measure(qlist, 100)
print(result)

```

C++

```

@settings:
    language = C++;
    autoimport = True;
    compile_only = False;

@qcodes:
variationalCircuit oneCircuit(vector<qubit> qlist, hamiltonian hp, avar beta, avar gamma)
↪{
    for(let i = 0 : 1: hp.size()){
        vector<qubit> tmp_vec;
        let hamiltonItem = hp[i];
        let dict_p = hamiltonItem.getFirst();
        for(map m in dict_p) {
            tmp_vec.append(qlist[m.first()]);
        }

        let coef = tmp_vec[1];
        VQG_CNOT(tmp_vec[0], tmp_vec[1]);
        VQG_RZ(tmp_vec[1], 2*gamma);
        VQG_CNOT(tmp_vec[0], tmp_vec[1]);
    }

    for(let j=0: 1: qlist.size()){
        VQG_RX(qlist[j], 2.0*beta);
    }
}

@script:

```

(continues on next page)

(continued from previous page)

```

void transverter(MatrixXf & friendship, PauliOperator::PauliMap & friendship_map) {
    for (size_t i = 0; i < friendship.rows(); i++) {
        for (size_t j = i+1; j < friendship.cols(); j++) {
            stringstream ss_temp;
            ss_temp << "Z" << i << " " << "Z" << j;
            string s_temp(ss_temp.str());
            friendship_map.insert(make_pair(s_temp, friendship(i, j)));
        }
    }

    cout << friendship.rows() << endl;
}

string testQAOA(MatrixXf & friendship) {
    auto qvm = initQuantumMachine(CPU);

    PauliOperator::PauliMap friendship_map;
    transverter(friendship, friendship_map);
    PauliOperator Hp(friendship_map);

    auto qubit_num = Hp.getMaxIndex();
    auto qvec = qAllocMany(qubit_num);
    int step = 4;

    var x(MatrixXd::Random(step, 1), true);
    var y(MatrixXd::Random(step, 1), true);

    VQC qprog;
    for (auto qubit : qvec) {
        qprog.insert(VQG_H(qubit));
    }

    for (size_t i = 0; i < step; i++) {
        qprog.insert(oneCircuit(qvec, Hp.toHamiltonian(), x[i], y[i]));
    }

    var loss = qop(qprog, Hp, qvm, qvec);
    auto optimizer = MomentumOptimizer::minimize(loss, 0.02, 0.9);
}

```

(continues on next page)

(continued from previous page)

```

    auto leaves = optimizer->get_variables();
    size_t iterations = 10;
    for (size_t i = 0; i < iterations; i++) {
        optimizer->run(leaves);
        std::cout << " iter: " << i << " loss : " << optimizer->get_loss() << std::endl;
    }

    QProg prog;
    prog << qprog.feed();
    directlyRun(prog);
    auto result = quickMeasure(qvec, 10);
    size_t temp = 0;
    string key;
    for (auto i : result) {
        if (i.second > temp) {
            temp = i.second;
            key = i.first;
        }
    }
    destroyQuantumMachine(qvm);
    return key;
}

int main() {
    const int prisoner_count = 4;
    MatrixXf friendship = MatrixXf::Zero(prisoner_count, prisoner_count);
    friendship << 0, 0.8, 0.2, -0.2,
                0.8, 0, 0, 0.7,
                0.2, 0, 0, -0.3,
                -0.2, 0.7, -0.3, 0;
    cout << friendship << endl;
    string result = testQAOA(friendship);
    cout << result << endl;
    getchar();
}

```

### 7.1.3 6.1.3 QAOA 算法小结

我们用于求解这些问题的经典方法已经历了数十年的打磨发展，效果已经相当好了。即使早期 NISQ 时代的量子设备还无法与最好的经典计算机媲美，实验结果也可能会激励我们期待看到 QAOA 或 VQE 在未来超

越经典方法，从而进一步推动技术发展。QAOA 很有意思的一个原因是它具有展示量子霸权潜力。

## 7.2 6.2 Deutsch–Jozsa 算法

### 7.2.1 6.2.1 Deutsch–Jozsa 算法介绍

Deutsch–Jozsa 算法是一种经过设计的情况，它证明了量子算法相对于经典算法有指数级别的加速能力。D-J 算法的问题描述是这样的：

**问题描述：**

考虑函数：

$$f : \{0, 1\}^n \rightarrow \{0, 1\}$$

我们保证有如下两种可能性：

- (1)  $f$  是常数的 (Constant)，即是对  $x \in \{0, 1\}^n$ ，都有  $f(x) = 0$  或  $f(x) = 1$ 。
- (2)  $f$  是平衡的 (Balanced)，对于输入的  $x \in \{0, 1\}^n$ ， $f(x)$  输出 0 和 1 的个数相同。

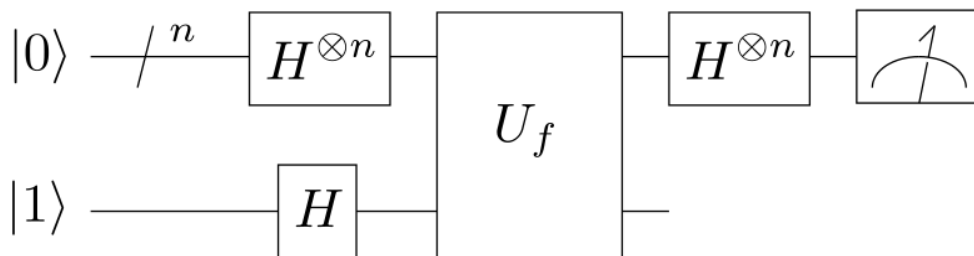
算法的目标：判断函数  $f$  是什么类型。

**经典算法情况：**

在最简单的情况下，最少也需要 2 次才能判断函数属于什么类型。因为需要第二个输出才能判断最终函数的类型。对于  $n$  位输入时，最坏的情况下需要  $2^{n-1}$  次才能确认。

**量子算法：**通过构造 Oracle 的方式，仅需运行一次就能确定函数属于哪一类。

**植入步骤：**



第一步，制备  $n$  个工作 (Working) 比特到  $|0\rangle$  态，与一个辅助 (Ancillary) 比特到  $|1\rangle$ 。

第二步，所有比特都经过 Hadamard 变换，使系统处于叠加态上。

$$|0\rangle^{\otimes n} |1\rangle \xrightarrow{H^{\otimes n+1}} \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right)$$

第三步，系统通过 Oracle，一种酉变换，满足：

$$U_f |x\rangle |y\rangle \rightarrow |x\rangle |y \oplus f(x)\rangle$$

这时候，系统状态为：

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle \left( \frac{(|0\rangle - |1\rangle)}{\sqrt{2}} \right) \xrightarrow{\text{oracle}} \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle \left( \frac{(|0\rangle - |1\rangle)}{\sqrt{2}} \right)$$

当  $f(x)=1$  时，会使得  $\frac{(|0\rangle - |1\rangle)}{\sqrt{2}} \rightarrow \frac{(|1\rangle - |0\rangle)}{\sqrt{2}}$ ，发生相位的翻转。

第四步：去除辅助比特，执行 Bell 测量。如果输出全部为 0，则是  $f$  是常数的，反之，这是平衡的。

## 7.2.2 6.2.2 Deutsch-Jozsa 算法的实现

下面给出 QRunes 实现 Deutsch-Jozsa 算法的代码示例：

Python

```
@settings:
    language = Python;
    autoimport = True;
    compile_only = False;

@qcodes:
circuit<vector<qubit>,qubit> generate_two_qubit_oracle(vector<bool> oracle_function){
    return lambda (vector<qubit> qlist,qubit qubit2):{
        if (oracle_function[0] == false &&
            oracle_function[1] == true)
        {
            // f(x) = x;
            CNOT(qlist[0], qubit2);
        }
        else if (oracle_function[0] == true &&
            oracle_function[1] == false)
        {
            // f(x) = x + 1;
            CNOT(qlist[0], qubit2);
            X(qubit2);
        }
        else if (oracle_function[0] == true &&
            oracle_function[1] == true)
        {
            // f(x) = 1
            X(qubit2);
        }
    }
    else
```

(continues on next page)

(continued from previous page)

```

        {
            // f(x) = 0, do nothing
        }
    };
}

Deutsch_Jozsa_algorithm(vector<qubit> qlist,qubit qubit2,vector<cbit> clist,circuit
↪<vector<qubit>,qubit> oracle){
    X(qubit2);
    apply_QGate(qlist, H);
    H(qubit2);
    oracle(qlist,qubit2);
    apply_QGate(qlist, H);
    measure_all(qlist,clist);
}

@script:
def two_qubit_deutsch_jozsa_algorithm(boolean_function):
    init(QMachineType.CPU_SINGLE_THREAD)
    qubit_num = 2
    cbit_num = 1
    qvec = qAlloc_many(qubit_num)
    cvec = cAlloc_many(cbit_num)
    oracle = generate_two_qubit_oracle(boolean_function)
    prog = Deutsch_Jozsa_algorithm([qvec[0]],qvec[1],[cvec[0]],oracle)
    result = directly_run(prog)
    if cvec[0].eval() == False:
        print("Constant function!")
    elif cvec[0].eval() == True:
        print("Balanced function!")
    finalize()

if __name__ == '__main__':
    fx0 = 0
    fx1 = 1
    print("input the input function")
    print("The function has a boolean input")
    print("and has a boolean output")
    print("f(0)= (0/1)?")

```

(continues on next page)

(continued from previous page)

```

fx0 = int(input())
print("f(1)=(0/1)?")
fx1 = int(input())
oracle_function = [fx0,fx1]
print("Programming the circuit...")
two_qubit_deutsch_jozsa_algorithm(oracle_function)

```

C++

```

@settings:
    language = C++;
    autoimport = True;
    compile_only = False;

@qcodes:
circuit<vector<qubit>,qubit> generate_two_qubit_oracle(vector<bool> oracle_function){
    return lambda (vector<qubit> qlist,qubit qubit2):{
        if (oracle_function[0] == false &&
            oracle_function[1] == true)
        {
            // f(x) = x;
            CNOT(qlist[0], qubit2);
        }
        else if (oracle_function[0] == true &&
            oracle_function[1] == false)
        {
            // f(x) = x + 1;
            CNOT(qlist[0], qubit2);
            X(qubit2);
        }
        else if (oracle_function[0] == true &&
            oracle_function[1] == true)
        {
            // f(x) = 1
            X(qubit2);
        }
        else
        {
            // f(x) = 0, do nothing
        }
    }
}

```

(continues on next page)



(continued from previous page)

```

    };
}

Deutsch_Jozsa_algorithm(vector<qubit> qlist,qubit qubit2,vector<cbit> clist,circuit
↪<vector<qubit>,qubit> oracle){
    X(qubit2);
    apply_QGate(qlist, H);
    H(qubit2);
    oracle(qlist,qubit2);
    apply_QGate(qlist, H);
    measure_all(qlist,clist);
}

@script:
void two_qubit_deutsch_jozsa_algorithm(vector<bool> boolean_function)
{
    init(QMachineType::CPU);
    auto qvec = qAllocMany(2);
    auto c = cAlloc();
    if (qvec.size() != 2)
    {
        QCERR("qvec size error the size of qvec must be 2");
        throw invalid_argument("qvec size error the size of qvec must be 2");
    }

    auto oracle = generate_two_qubit_oracle(boolean_function);
    QProg prog;
    prog << Deutsch_Jozsa_algorithm({ qvec[0] }, qvec[1], { c }, oracle);

    /* To Print The Circuit */
    /*
    extern QuantumMachine* global_quantum_machine;
    cout << transformQProgToQRunes(prog, global_quantum_machine) << endl;
    */

    directlyRun(prog);
    if (c.eval() == false)
    {
        cout << "Constant function!" << endl;
    }
}

```

(continues on next page)

```

        else if (c.eval() == true)
        {
            cout << "Balanced function!" << endl;
        }
        finalize();
    }

    int main() {
        bool fx0 = 0, fx1 = 0;
        cout << "input the input function" << endl
            << "The function has a boolean input" << endl
            << "and has a boolean output" << endl
            << "f(0)= (0/1)?";
        cin >> fx0;
        cout << "f(1)=(0/1)?";
        cin >> fx1;
        std::vector<bool> oracle_function({ fx0,fx1 });
        cout << "Programming the circuit..." << endl;
        two_qubit_deutsch_jozsa_algorithm(oracle_function);
    }

```

### 7.2.3 6.2.3 Deutsch–Jozsa 算法小结

经典算法的验证次数是  $O(2^n)$  的，量子算法算上叠加态的准备和测量的时间，需要的操作步骤为  $O(n)$ 。所以我们说明量子算法相对于经典算法具有指数级别加速的特性。D-J 算法的问题在于它解决的问题既不实用，又具有很大的限制（要求平衡函数中必须恰好为一半 0 一半 1）。另外，我们还对黑盒子本身的形态有要求。所以说 D-J 算法的理论意义是远大于其实用意义的。

## 7.3 6.3 Grover 算法

### 7.3.1 6.3.1 Grover 算法介绍

什么是搜索算法呢？举一个简单的例子，在下班的高峰期，我们要从公司回到家里。开车走怎样的路线才能够耗时最短呢？我们最简单的想法，当然是把所有可能的路线一次一次的计算，根据路况计算每条路线所消耗的时间，最终可以得到用时最短的路线，即为我们要找的最快路线。这样依次将每一种路线计算出来，最终对比得到最短路线。搜索的速度与总路线数  $N$  相关，记为  $O(N)$ 。而采用量子搜索算法，则可以以  $O(\sqrt{N})$  的速度进行搜索，要远快于传统的搜索算法。

那么我们怎么实现 Grover 搜索算法呢？

首先，我们先化简一下搜索模型。我们将所有数据存在数据库中，假设我们有  $n$  个量子比特，用来记录数据库中的每一个数据的索引，一共可以表示  $2^n$  个数据，记为  $N$  个。我们希望搜索得到的数据有  $M$  个。为了表示一个数据是否是我们搜索的结果。我们建立一个函数：

$$f(x) = \begin{cases} 0 & (x \neq x_0) \\ 1 & (x = x_0) \end{cases}$$

其中  $x_0$  为我们的搜索目标的索引值。也就是说，当我们搜索到我们的目标时，我们的函数值  $f(x)$  置为 1，如果搜索的结果不是我们的目标时， $f(x)$  置为 0。

接下来，我们假设有一个量子 Oracle 可以识别搜索问题的解，是别的结果通过 Oracle 的一个量子比特给出。我们可以将 Oracle 定义为：

$$|x\rangle|q\rangle \xrightarrow{\text{Oracle}} |x\rangle|q + f(x)\rangle$$

其中  $|q\rangle$  是一个结果寄存器，是二进制加法，通过 Oracle，我们可以实现，当搜索的索引为我们的目标结果时，结果寄存器翻转；反之，结果寄存器值不变。从而我们可以通过判断结果寄存器的值，来确定搜索的对象是否为我们目标值。如此描述 Oracle 有些抽象，Oracle 对量子态的具体操作是什么样的呢？同 D-J 算法相似，我们先将初态制备在  $|0\rangle^{\otimes n}|1\rangle$  态上， $|0\rangle^{\otimes n}$  为查询寄存器， $|1\rangle$  为结果寄存器。经过 Hadamard 门操作后，可以将查询寄存器的量子态，变为所有结果的叠加态。换句话说，经过了 Hadamard 门，我们就可以得到所有结果的索引。而结果寄存器则变为  $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$  接下来，使其通过 Oracle，可以对每一个索引都进行一次检验，如果是我们的目标结果，则将答案寄存器的量子态进行 0、1 翻转，即答案寄存器变为

$$\frac{1}{\sqrt{2}}(|1\rangle - |0\rangle) = -\frac{1}{\sqrt{2}}(|1\rangle - |0\rangle)$$

，而查询寄存器不变。而当检验的索引不是我们要求的结果时，寄存器均不发生改变。因此，Oracle 可以换一种表示方式

$$|x\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \xrightarrow{\text{Oracle}} (-1)^{f(x)} |x\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right)$$

其中， $|x\rangle$  是查询寄存器的等额叠加态中的一种情况。

也就是说，Oracle 的作用，是通过改变了解的相位，标记了搜索问题的解。

现在，我们已经将搜索问题的解通过相位标记区分出来了。那么如何能够将量子态的末态变为已标记出的态呢？

我们将问题换一种思路进行考虑。我们知道，当查询寄存器由初态经过 Hadamard 门后，会变为所有可能情况的等额叠加态。也就是说，它包含着所有搜索问题的解与非搜索问题的解。我们将这个态记为

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_x |x\rangle$$

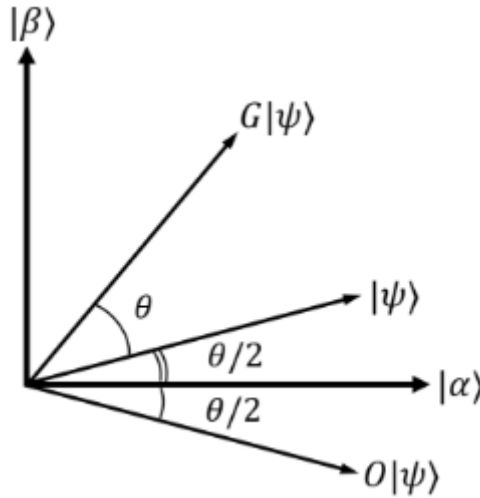
我们将所有非搜索问题的解定义为一个量子态  $|\alpha\rangle$ ，其中  $\sum_{x_1}$  代表着  $x$  上所有非搜索问题的解的和。

$$|\alpha\rangle = \frac{1}{\sqrt{N-M}} \sum_{x_1} |x\rangle$$

显然， $|\beta\rangle$  为我们期望的最终量子态，而且  $|\alpha\rangle$  和  $|\beta\rangle$  相互正交。利用简单的代数运算，我们就可以将初态  $|\psi\rangle$  重新表示为

$$|\psi\rangle = \sqrt{\frac{N-M}{N}} |\alpha\rangle + \sqrt{\frac{M}{N}} |\beta\rangle$$

也就是说，我们用搜索问题的解的集合和非搜索问题的解的集合，重新定义了初始态换句话说，我们的初态属于  $|\alpha\rangle$  与  $|\beta\rangle$  张成的空间。因此，我们可以用平面向量来表示这三个量子态，如图。



那么，Oracle 作用在新的表示方法下的初态会产生怎样的影响呢？我们知道，Oracle 的作用是用负号标记搜索问题的解，因此相当于将  $|\beta\rangle$  内每一个态前均增加一个负号，将所有的负号提取出来，可以得到：

$$|\psi\rangle \xrightarrow{\text{Oracle}} \sqrt{\frac{N-M}{N}}|\alpha\rangle - \sqrt{\frac{M}{N}}|\beta\rangle$$

对应在平面向量中，相当于将  $|\psi\rangle$  做关于  $|\alpha\rangle$  轴的对称。但是，仅仅有这一种操作，是无法将量子态从  $|\psi\rangle$  变为  $|\beta\rangle$ 。我们还需要另一种对称操作。

第二种对称操作，是将量子态关于  $|\psi\rangle$  对称的操作。这个操作由三个部分构成。

1. 将量子态经过一个 Hardmard 门。
2. 对量子态进行一个相位变换，将  $|0\rangle^n$  态的系数保持不变，将其他的量子态的系数增加一个负号。相当于  $2|0\rangle\langle 0| - I$  酉变换算子。
3. 再经过一个 Hardmard 门。

这三步操作的数学表述为：

$$H^{\otimes n}(2|0\rangle\langle 0| - I)H^{\otimes n} = 2|\psi\rangle\langle\psi| - I$$

上述过程涉及到复杂的量子力学知识，如果你不理解，没关系。你只需要知道，这三部分的操作，只是为了实现将量子态关于  $|\psi\rangle$  对称即可。如果你想了解为什么这三步操作可以实现，可以阅读关于量子计算相关书籍进一步理解。

前面介绍的两种对称操作，合在一起称为一次 Grover 迭代。假设初态  $|\psi\rangle$  与  $|\alpha\rangle$  可以表示为

$$|\psi\rangle = \cos\frac{\theta}{2}|\alpha\rangle + \sin\frac{\theta}{2}|\beta\rangle$$

很容易得到

$$\cos\frac{\theta}{2} = \sqrt{\frac{N-M}{N}}$$

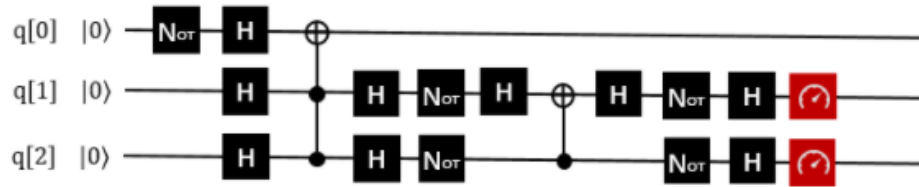
可以从几何图像上看到，每一次 Grover 迭代，可以使量子态逆时针旋转  $\theta$ 。经历了  $k$  次 Grover 迭代，末态的量子态为：

$$G^k|\psi\rangle = \cos\left(\frac{2k+1}{2}\theta\right)|\alpha\rangle + \sin\left(\frac{2k+1}{2}\theta\right)|\beta\rangle$$

因此，经过多次迭代操作，总可以使末态在  $|\beta\rangle$  态上概率很大，满足精确度的要求。经过严格的数学推导，可证明，迭代的次数  $R$  满足：

$$R \leq \frac{\pi}{4} \sqrt{\frac{N}{M}}$$

参考线路图：



### 7.3.2 6.3.2 Grover 算法的实现

下面给出 QRunes 实现 Grover 算法的代码示例：

Python

```
@settings:
    language = Python;
    autoimport = True;
    compile_only = False;

@qcodes:
circuit<vector<qubit>,qubit> generate_3_qubit_oracle(int target){
    return lambda (vector<qubit> qvec,qubit qu):{
        if(target == 0){
            X(qvec[0]);
            X(qvec[1]);
            Toffoli(qvec[0], qvec[1], qu);
            X(qvec[0]);
            X(qvec[1]);
        }
        if(target == 1){
            X(qvec[0]);
            Toffoli(qvec[0], qvec[1], qu);
            X(qvec[0]);
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
    if(target == 2){
        X(qvec[1]);
        Toffoli(qvec[0], qvec[1], qu);
        X(qvec[1]);
    }
    if(target == 3){
        Toffoli(qvec[0], qvec[1], qu);
    }
};
}

circuit diffusion_operator(vector<qubit> qvec){
    vector<qubit> controller;
    controller = qvec[0:qvec.length()-1];
    apply_QGate(qvec, H);
    apply_QGate(qvec, X);
    Z(qvec[qvec.length()-1]).control(controller);
    apply_QGate(qvec, X);
    apply_QGate(qvec, H);
}

Grover_algorithm(vector<qubit> working_qubit, qubit ancilla, vector<cbit> cvec,
                 circuit<vector<qubit>,qubit> oracle, int repeate){

    X(ancilla);
    apply_QGate(working_qubit, H);
    H(ancilla);

    if(repeate == 0){
        let sqrtN = 1 << (working_qubit / 2);
        repeate = 100 * sqrtN;
    }

    for(let i = 0 : 1 : repeate){
        oracle(working_qubit,ancilla);
        diffusion_operator(working_qubit);
    }

    measure_all(working_qubit,cvec);

```

(continues on next page)

(continued from previous page)

```

}

@script:
if __name__ == '__main__':
    condition = 1
    while condition == 1:
        print("input the input function")
        print("The function has a boolean input")
        print("and has a boolean output")
        print("target=(0/1/2/3)?")
        target = int(input())
        print("Programming the circuit...")
        oracle = generate_3_qubit_oracle(target)

        qvm = init_quantum_machine(QMachineType.CPU_SINGLE_THREAD)

        qubit_number = 3

        working_qubit = qvm.qAlloc_many(qubit_number-1)

        ancilla = qvm.qAlloc()

        cbitnum = 2
        cvec = qvm.cAlloc_many(cbitnum)

        repeate = 1

        prog = Grover_algorithm(working_qubit, ancilla, cvec, oracle, repeate)

        # To Print The Circuit
        print(to_QRunes(prog, qvm))

        resultMap = directly_run(prog)

        if resultMap["c0"]:
            if resultMap["c1"]:
                print("target number is 3 !")
            else:
                print("target number is 2 !")
        else:

```

(continues on next page)

(continued from previous page)

```

    if resultMap["c1"]:
        print("target number is 1 !")
    else:
        print("target number is 0 !")
destroy_quantum_machine(qvm)

```

C++

```

@settings:
    language = C++;
    autoimport = True;
    compile_only = False;

@qcodes:
circuit<vector<qubit>,qubit> generate_3_qubit_oracle(int target){
    return lambda (vector<qubit> qvec,qubit qu):{
        if(target == 0){
            X(qvec[0]);
            X(qvec[1]);
            Toffoli(qvec[0], qvec[1], qu);
            X(qvec[0]);
            X(qvec[1]);
        }
        if(target == 1){
            X(qvec[0]);
            Toffoli(qvec[0], qvec[1], qu);
            X(qvec[0]);
        }
        if(target == 2){
            X(qvec[1]);
            Toffoli(qvec[0], qvec[1], qu);
            X(qvec[1]);
        }
        if(target == 3){
            Toffoli(qvec[0], qvec[1], qu);
        }
    };
}

circuit diffusion_operator(vector<qubit> qvec){

```

(continues on next page)



(continued from previous page)

```

    vector<qubit> controller;
    controller = qvec[0:qvec.length()-1];
    apply_QGate(qvec, H);
    apply_QGate(qvec, X);
    Z(qvec[qvec.length()-1]).control(controller);
    apply_QGate(qvec, X);
    apply_QGate(qvec, H);
}

Grover_algorithm(vector<qubit> working_qubit, qubit ancilla, vector<cbit> cvec,
                 circuit<vector<qubit>,qubit> oracle, int repeate){

    X(ancilla);
    apply_QGate(working_qubit, H);
    H(ancilla);

    if(repeate == 0){
        let sqrtN = 1 << (working_qubit.size() / 2);
        repeate = 100 * sqrtN;
    }

    for(let i = 0 : 1 : repeate){
        oracle(working_qubit,ancilla);
        diffusion_operator(working_qubit);
    }

    measure_all(working_qubit,cvec);
}

@script:
int main()
{
    while (1) {
        int target;
        cout << "input the input function" << endl
             << "The function has a boolean input" << endl
             << "and has a boolean output" << endl
             << "target=(0/1/2/3)?";
        cin >> target;
    }
}

```

(continues on next page)

(continued from previous page)

```
cout << "Programming the oracle..." << endl;
Oracle<QVec, Qubit*> oracle = generate_3_qubit_oracle(target);

init(QMachineType::CPU_SINGLE_THREAD);

int qubit_number = 3;
vector<Qubit*> working_qubit = qAllocMany(qubit_number - 1);
Qubit* ancilla = qAlloc();

int cbitnum = 2;
vector<ClassicalCondition> cvec = cAllocMany(cbitnum);

auto prog = Grover_algorithm(working_qubit, ancilla, cvec, oracle, 1);

/* To Print The Circuit */

extern QuantumMachine* global_quantum_machine;
cout << transformQProgToQRunes(prog, global_quantum_machine) << endl;

auto resultMap = directlyRun(prog);
if (resultMap["c0"])
{
    if (resultMap["c1"])
    {
        cout << "target number is 3 !";
    }
    else
    {
        cout << "target number is 2 !";
    }
}
else
{
    if (resultMap["c1"])
    {
        cout << "target number is 1 !";
    }
    else
    {
```

(continues on next page)

(continued from previous page)

```

        cout << "target number is 0 !";
    }
}
finalize();
}
return 0;
}

```

### 7.3.3 6.3.3 Grover 算法小结

1996 年, Lov Grover 提出了量子搜索算法, 对于  $N$  个无序列数据里寻求 1 个有效数据, 经典算法给出的有效时间复杂度为  $O(N)$ , 而 Grover 证明了处理同样的问题, 量子算法可以做到时间复杂度为  $O(\sqrt{N})$ 。也就是说 Grover 的搜索算法可以以指数级的加速改善搜索复杂度。如何更直观理解: 假设给定相同的问题, 量子计算用 10000 次就解决, 但是经典计算机则需要  $10000^2=100000000$ , 这是一万和一亿的差距。由此可见, 对于大数据的搜索, Grover 算法印证了量子计算能大显身手, 可有效解决搜索问题。

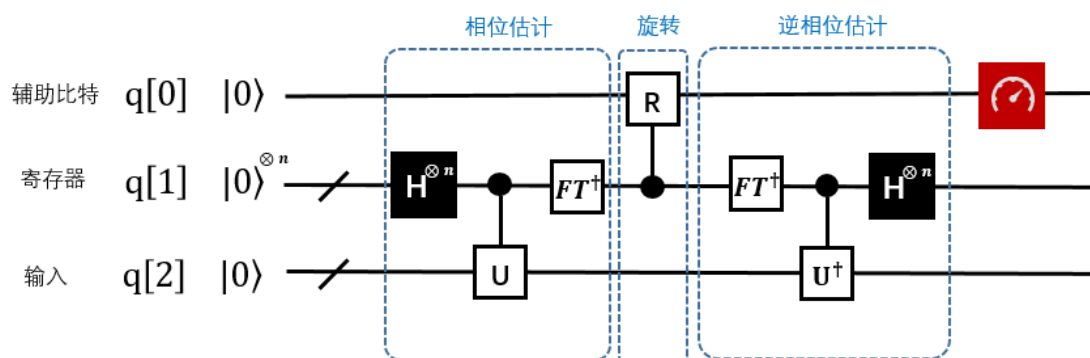
## 7.4 6.4 HHL 算法

### 7.4.1 6.4.1 HHL 算法介绍

HHL 算法是一个用量子计算机解决线性问题  $Ax=b$  最优解的算法, 广泛的被应用于许多量子机器学习算法中 (如支持向量机 SVM, 主成分分析 PCA 等等)。量子算法在其经典计算对比下, 呈指数级加速。Harrow, Hassidim 和 Lloyd (HHL) 提出了一种求解线性系统  $Ax=b$  (其中  $A$  是算子,  $x, b$  是向量) 中  $x$  信息的量子线性系统分析。HHL 算法解决了什么样的问题? 那就是求解线性方程的问题。

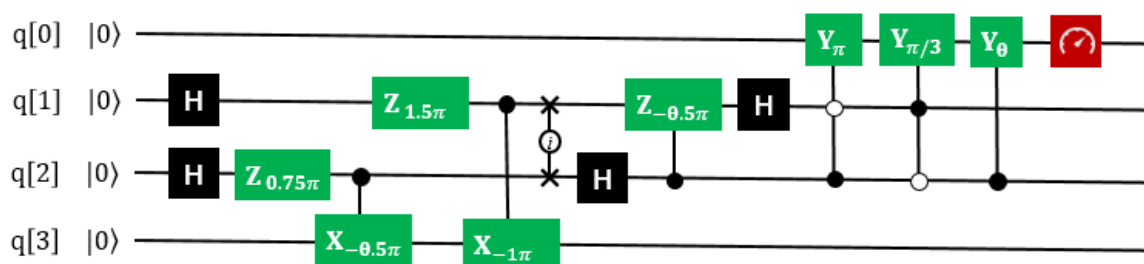
HHL 算法的输入和输出:

- 输入: 一个  $n \times n$  的矩阵  $A$  和一个  $n$  维向量  $b$ ,
- 输出:  $n$  维向量  $x$ , 满足  $Ax=b$ 。



HHL 的限制条件:

1. 输入的矩阵, 必须是 adjoint 矩阵, 当  $A$  不是 Hermitian 时, 需要构造 adjoint 矩阵。算法的输入部分如图 1 中红色方框所标出。输入  $q[2]$  存放在底部寄存器中, 输入  $A$  作为相位估计中酉算子的一个组成部分。
2. 输出  $x$  的形式: 算法的输出如红色部分标出 (同一个寄存器)。底部寄存器存放的是一个蕴含了向量  $x$  的量子态。此处不需要知道这个状态具体情况。



## 7.4.2 6.4.2 HHL 算法的实现

下面给出 QRunes 实现 HHL 算法的代码示例:

Python

```
@settings:
    language = Python;
    autoimport = True;
    compile_only = False;

@qcodes:
circuit CRotate(vector<qubit> q) {
    vector<qubit> controlVector;
    controlVector.append(q[1]);
    controlVector.append(q[2]);
    X(q[1]);
    RY(q[0], Pi).control(controlVector);
    X(q[1]);
    X(q[2]);
    RY(q[0], Pi/3).control(controlVector);
    X(q[2]);
    RY(q[0], 0.679673818908).control(controlVector); //arcsin(1/3)
}
```

(continues on next page)

(continued from previous page)

```

//Phase estimation algorithms
circuit hhlPse(vector<qubit> q) {

    H(q[1]);
    H(q[2]);
    RZ(q[2], 0.75*Pi);
    CU(Pi, 1.5*Pi, -0.5*Pi, Pi/2, q[2], q[3]);
    RZ(q[1], 1.5*Pi);
    CU(Pi, 1.5*Pi, -Pi, Pi/2, q[1], q[3]);

    CNOT(q[1], q[2]);
    CNOT(q[2], q[1]);
    CNOT(q[1], q[2]);

    H(q[2]);
    CU(-0.25*Pi, -0.5*Pi, 0, 0, q[2], q[1]);
    H(q[1]);
}

hhl_no_measure(vector<qubit> qlist, vector<cbit> clist) {
    //phase estimation
    hhlPse(qlist);
    //rotate
    CRotate(qlist);
    measure(qlist[0], clist[0]);
    qif (clist[0]) {
        hhlPse(qlist).dagger();
    }
}

@script:
if __name__ == '__main__':
    init(QMachineType.CPU_SINGLE_THREAD)

    qubit_num = 4
    cbit_num = 2
    qv = qAlloc_many(qubit_num)
    cv = cAlloc_many(cbit_num)
    hhlprog = QProg()
    hhlprog.insert(RY(qv[3], 3.14159265358979/2))    #change vecotr b in equation Ax=b

```

(continues on next page)

(continued from previous page)

```

hhlprog.insert(hhl_no_measure(qv, cv))
directly_run(hhlprog)
pmeas_q = []
pmeas_q.append(qv[3])
res = PMeasure_no_index(pmeas_q)
print('prob0: %s' %(res[0]))
print('prob1: %s' %(res[1]))

finalize()

```

C++

```

@settings:
    language = C++;
    autoimport = True;
    compile_only = False;

@qcodes:
circuit CRotate(vector<qubit> q) {
    vector<qubit> controlVector;
    controlVector.append(q[1]);
    controlVector.append(q[2]);
    X(q[1]);
    RY(q[0], Pi).control(controlVector);
    X(q[1]);
    X(q[2]);
    RY(q[0], Pi/3).control(controlVector);
    X(q[2]);
    RY(q[0], 0.679673818908).control(controlVector); //arcsin(1/3)
}

//Phase estimation algorithms
circuit hhlPse(vector<qubit> q) {

    H(q[1]);
    H(q[2]);
    RZ(q[2], 0.75*Pi);
    CU(Pi, 1.5*Pi, -0.5*Pi, Pi/2, q[2], q[3]);
    RZ(q[1], 1.5*Pi);
    CU(Pi, 1.5*Pi, -Pi, Pi/2, q[1], q[3]);

```

(continues on next page)

(continued from previous page)

```

    CNOT(q[1], q[2]);
    CNOT(q[2], q[1]);
    CNOT(q[1], q[2]);

    H(q[2]);
    CU(-0.25*Pi, -0.5*Pi, 0, 0, q[2], q[1]);
    H(q[1]);
}

hhl_no_measure(vector<qubit> qlist, vector<cbit> clist) {
    //phase estimation
    hhlPse(qlist);
    //rotate
    CRotate(qlist);
    measure(qlist[0], clist[0]);
    qif (clist[0]) {
        hhlPse(qlist).dagger();
    }
}

@script:
int main() {
    map<string, bool> temp;
    int x0 = 0;
    int x1 = 0;

    init(QMachineType::CPU);
    int qubit_number = 4;
    vector<Qubit*> qv = qAllocMany(qubit_number);
    int cbitnum = 2;
    vector<ClassicalCondition> cv = cAllocMany(2);

    auto hhlprog = CreateEmptyQProg();
    hhlprog << RY(qv[3], PI / 2); // change vecotr b in equation Ax=b
    hhlprog << hhl_no_measure(qv, cv);
    directlyRun(hhlprog);
    QVec pmeas_q;
    pmeas_q.push_back(qv[3]);
    vector<double> s = PMeasure_no_index(pmeas_q);
}

```

(continues on next page)

(continued from previous page)

```
cout << "prob0:" << s[0] << endl;
cout << "prob1:" << s[1] << endl;
finalize();
}
```

### 7.4.3 6.4.3 HHL 算法小结

线性系统是很多科学和工程领域的核心，由于 HHL 算法在特定条件下实现了相较于经典算法有指数加速效果，从而未来能够在机器学习、数值计算等场景有优势体现。配合 Grover 算法在数据方面的加速，将是未来量子机器学习，人工智等科技得以突破的关键性技术。

## 7.5 6.5 QuantumWalk 算法

### 7.5.1 6.5.1 QuantumWalk 算法介绍

QuantumWalk 作为一种新的量子计算模型具有巨大的前景，为经典算法寻求量子版本的解决方案提供了新思路。将量子漫步理论与聚类算法相结合，在分析图上离散量子漫步特点及其在解决聚类问题时存在不足的前提下，采取将漫步空间网格化的方式将模型简化，提出一种网格化量子漫步聚类模型，使之能够很好的完成聚类任务，该模型将数据点考虑为在量子网格世界中的根据特定规则执行漫步过程的量子，由于量子叠加等特性的存在，量子漫步聚类居右更好的时间效率和侦探能力，仿真实验也表明算法在聚类正确性上具有不错的表现。

近十年来，量子漫步作为一种新的量子计算模型崭露头角，并由于量子漫步构造的量子算法在许多问题的求解上相比于经典算法具有明显的优势，因此其在搜索、组合优化、元素区分等领域均取得了重大的进展。另外，Childs 和 Lovett 等分别提出了离散和连续两种具有通用意义上的量子漫步架构，阐述了一切量子算法均可在建立于量子漫步模型的一般算法框架，这促成了量子漫步模型成为构建通用算法的新思路。

### 7.5.2 6.5.2 quantumWalk 算法的实现

下面给出 QRunes 实现 QuantumWalk 算法的代码示例：

Python

```
@settings:
    language = Python;
    autoimport = True;
    compile_only = False;

@qcodes:
```

(continues on next page)



(continued from previous page)

```

circuit addOne(vector<qubit> q) {
    vector<qubit> vControlQubit;
    vControlQubit = q[1:q.length()-1];
    for (let i=0: 1: q.length()) {
        X(q[i]).control(vControlQubit);
        if (vControlQubit.length() >= 1) {
            vControlQubit.pop(0);
        }
    }
}

circuit walkOneStep(vector<qubit> qvec) {
    let iLength = qvec.length();
    X(qvec[iLength-1]);
    vector<qubit> vControlQbit;
    vControlQbit = qvec[1:qvec.length()];
    circuit qCircuit1 = addOne(qvec);
    circuit qCircuit2 = addOne(qvec);
    X(qvec[iLength-1]);
    qCircuit2.dagger();
}

//continuous quantum walks,consists of a walker and an evolution operator.
quantumWalk(vector<qubit> q, vector<cbit> c) {
    let length = q.length();
    X(q[length-2]);
    X(q[length-2]);
    for (let i=0: ((1 << length)-1): 1) {
        H(q[length - 1]);
        walkOneStep(q);
    }
}

@script:
import sys
if __name__ == '__main__':
    print('welcome to Quantum walk')
    qubit_num = int(input('please input qubit num\n'))
    if qubit_num < 0 or qubit_num > 24:
        print('error: qubitnum need > 0 and < 24')

```

(continues on next page)

(continued from previous page)

```

    sys.exit(1)
init(QMachineType.CPU_SINGLE_THREAD)

cbit_num = qubit_num
qv = qAlloc_many(qubit_num)
cv = cAlloc_many(cbit_num)
qv.append(qAlloc())
qwAlgorithm = quantumWalk(qv, cv)
result = prob_run_dict(qwAlgorithm, qv)
for key,value in result.items():
    print(str(key) + " : " + str(value))

finalize()

```

C++

```

@settings:
    language = C++;
    autoimport = True;
    compile_only = False;

@qcodes:
circuit addOne(vector<qubit> q) {
    vector<qubit> vControlQubit;
    vControlQubit = q[1:q.length()-1];
    for (let i=0: 1: q.length()) {
        X(q[i]).control(vControlQubit);
        if (vControlQubit.length() >= 1) {
            vControlQubit.remove(0);
        }
    }
}

circuit walkOneStep(vector<qubit> qvec) {
    let iLength = qvec.length();
    X(qvec[iLength-1]);
    vector<qubit> vControlQbit;
    vControlQbit = qvec[1:qvec.length()];
    circuit qCircuit1 = addOne(qvec);
    circuit qCircuit2 = addOne(qvec);
}

```

(continues on next page)

(continued from previous page)

```

        X(qvec[iLength-1]);
        qCircuit2.dagger();
    }

//continuous quantum walks,consists of a walker and an evolution operator.
quantumWalk(vector<qubit> q, vector<cbit> c) {
    let length = q.length();
    X(q[length-2]);
    X(q[length-2]);
    for (let i=0: ((1 << length)-1): 1) {
        H(q[length - 1]);
        walkOneStep(q);
    }
}

@script:
int main() {
    int qubitnum = 0;
    cout << "welcome to Quantum walk\n" << endl
        << "\n" << endl
        << "please input qubit num\n";
    cin >> qubitnum;

    if ((qubitnum < 0) || (qubitnum > 24))
    {
        QCERR("qubitnum need > 0 and <24");
        exit(1);
    }
    init(QMachineType::CPU);
    vector<Qubit*> qVec = qAllocMany(qubitnum);
    vector<ClassicalCondition> cVec = cAllocMany(qubitnum);
    qVec.push_back(qAlloc());
    auto qwAlgorithm = quantumWalk(qVec, cVec);
    auto reuslt = directlyRun(qwAlgorithm);

    for(auto var : reuslt)
    {
        cout << var.first << " : " << var.second << endl;
    }
    finalize();
}

```

(continues on next page)

}

### 7.5.3 6.5.3 QuantumWalk 算法小结

量子漫步是一种典型的量子计算模型, 近年来开始受到量子计算理论研究者们的广泛关注。该算法的时间复杂度与 Grover 算法相同, 但是当搜索的目标数目多于总数的  $1/3$  时搜索成功概率大于 Grover 算法。量子漫步的实现对于研发量子计算机具有开创性的重大意思, 通过它新的算法就可以得到应用。比如, 在现在技术中, 要从一串 0 中找到某一个 0, 人们必须检查每个数位, 所需的时间随 0 的总体数量的增加而线性增加。如果使用量子漫步算法, 漫步者可以同时在一处搜索, “大海捞针” 的速度就被极大的提高了

## 7.6 6.6 Simon 算法

### 7.6.1 6.6.1 Simon 算法介绍

Simon 问题是 Daniel Simon 在 1994 年提出。它是一个计算问题, 可以在量子计算机上以指数速度相较经典计算机更快地解决。虽然这个问题本身目前没有在在实际应用中产生实际价值, 但它的趣味内涵在于它证明了量子算法可以比任何经典算法更快地解决这个问题。

Simon 的算法也启发了 Shor 算法 (详细请查阅 Shor 量子算法)。这两个问题都是阿贝尔隐子群 (Abelian hidden subgroup problem) 问题的特例, 而且是现今已知有效的量子算法。

#### 问题描述:

- 给定一个方程:

$$f : \{0, 1\}^n \rightarrow \{0, 1\}^n$$

- 存在  $s \in \{0, 1\}^n$ , 对所有的  $x, y \in \{0, 1\}^n$ , 满足下面的性质:  $f(x) = f(y)$  当且仅当  $x = y$  或  $x \oplus y = s$

(这里  $\oplus$  表示模 2 加。)

#### 算法目标:

寻找  $s$ 。

例:  $n=2$  的 Simon 问题, 考虑 2 量子比特。注意, 如果目标  $s = 0^n$ , 那这个函数是 1 对 1 (one-to-one) 的, 此处不考虑。反之, 则是一个二对一 (two-to-one) 的函数, 几种情况如下图 (函数值任意给定):

| (1)s=01 |      | (2)s=10 |      | (3)s=11 |      |
|---------|------|---------|------|---------|------|
| x       | f(x) | x       | f(x) | x       | f(x) |
| 00      | 1    | 00      | 1    | 00      | 1    |
| 01      | 1    | 01      | 2    | 01      | 3    |
| 10      | 0    | 10      | 1    | 10      | 3    |
| 11      | 0    | 11      | 2    | 11      | 1    |

在 (1) 很容易看出  $f(00) = f(01) = 1$ ,  $f(10) = f(11) = 0$ , 因此  $00\ 01 = 01$ ,  $10\ 11 = 01$ , 推出  $s = 01$ 。经典算法最低需要 2 次的才能确定, 一般情况下, 对于  $n$  比特的的问题估计找到目标  $s$  最糟糕的情况下要消耗多达  $2^{n-1} + 1$  次。但是在量子算法里, 1 次就解决了这个问题。

## 量子 Oracle

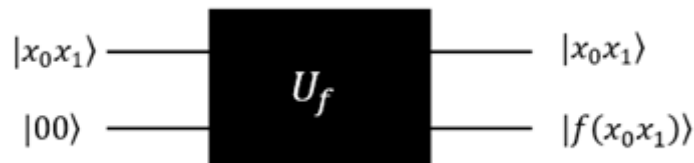
Simon 问题的量子 Oracle(考虑  $s=11$ )

考虑  $n=2$  的 Simon 问题, 此时需要 2 量子比特的变量和 2 量子比特的函数, 合计需要 4 量子比特。

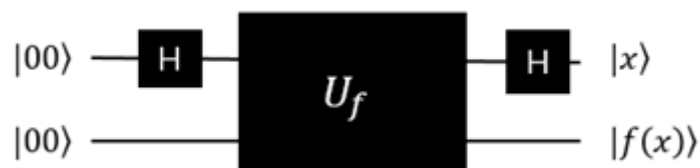
- 下面考虑 Simon 问题的 Oracle:

$$|x_0x_1\rangle|00\rangle \xrightarrow{U_f} |x_0x_1\rangle|f(x_0x_1)\rangle = |x_0x_1\rangle|f(x_0x_1)\rangle$$

- 线路图如下:



上面的这个量子 Oracle 可以加入 Hadamard 门, 对前两个量子比特做 H 操作, 等价于:



过程：

$$|0000\rangle \xrightarrow{HHII} |++\rangle |00\rangle \xrightarrow{U_f} \frac{1}{2}[(|00\rangle + |11\rangle)|1\rangle + (|01\rangle + |10\rangle)|3\rangle] \\ \xrightarrow{HHII} \frac{1}{2}[(|00\rangle + |11\rangle)|1\rangle + (|00\rangle - |11\rangle)|3\rangle]$$

(注意：|3 是我定义的函数值)

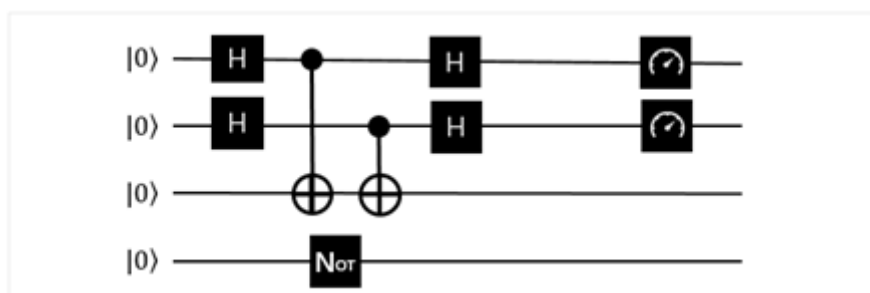
因此，最下面的两个位分别对应了 |1 和 |3，测量上面的两量子位，|00 和 |11 则会被以 50% 的概率被观察到。

下面是 QRunes 的实施过程：

1. 初始化 4 个量子比特。
2. 创建线路图：q[0]，q[1] 分别做 Hadamard 操作。
3. 对 q[0]，q[2] 和 q[1]，q[2] 分别执行 CNOT 操作。
4. 对 q[3] 执行 NOT 操作。
5. 再对 q[0]，q[1] 分别做 Hadamard 操作
6. 最后测量全部量子逻辑位，输出结果。

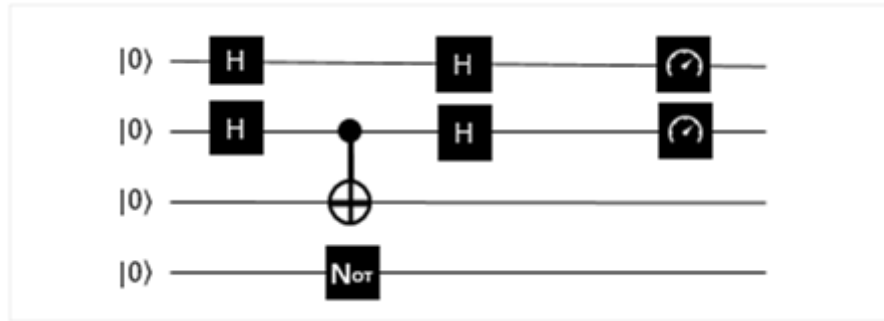
Simon 问题 (s=11) 的线路图设计参考图：

这里，测定结果得 |00 的时候，表示没有得到任何的信息，当测量得到 |11 的时候，就得到了 s=11，也就是说 Simon 量子算法里面，0 以外的获取 s 的概率为 50%。



s=10 的线路图参考，流程和思路和上面完全一致，测试用。

s= 10:



### 7.6.2 6.6.2 Simon 算法的实现

下面给出 QRunes 实现 Simon 算法的代码示例：

Python

```
@settings:
    language = Python;
    autoimport = True;
    compile_only = False;

@qcodes:
circuit controlfunc(vector<qubit> qvec, int index, int value) {
    let cd = qvec.length() / 2;
    vector<qubit> qvtemp;
    qvtemp = qvec[0:cd];
    if (index == 1) {
        X(qvec[0]);
    } else if (index == 2) {
        X(qvec[1]);
    } else if (index == 0) {
        X(qvec[0]);
        X(qvec[1]);
    }

    if (value == 1) {
        X(qvec[3]).control(qvtemp);
    } else if (value == 2) {
        X(qvec[2]).control(qvtemp);
    } else if (value == 3) {
        X(qvec[2]).control(qvtemp);
        X(qvec[3]).control(qvtemp);
    }
}
```

(continues on next page)

(continued from previous page)

```

    }

    if (index == 1) {
        X(qvec[0]);
    } else if (index == 2) {
        X(qvec[1]);
    } else if (index == 0) {
        X(qvec[0]);
        X(qvec[1]);
    }
}

circuit oraclefunc(vector<qubit> qvec, vector<int> funvalue) {
    let cd = qvec.length()/2;
    for (let i=0: 1: 4){
        let value = funvalue[i];
        controlfunc(qvec, i, value);
    }
}

Simon_QProg(vector<qubit> qvec, vector<cbit> cvec, vector<int> funvalue) {
    let cd = cvec.length();
    for (let i=0: 1: cd) {
        H(qvec[i]);
    }
    oraclefunc(qvec, funvalue);
    for (let i=0: 1: cd) {
        H(qvec[i]);
    }
    for (let i=0: 1: cd) {
        measure(qvec[i], cvec[i]);
    }
}

@script:
if __name__ == '__main__':
    print('4-qubit Simon Algorithm')
    print('f(x)=f(y)\t x+y=s')
    print('input f(x),f(x):[0,3]')
    func_value = []

```

(continues on next page)



(continued from previous page)

```

func_value.append(int(input('input f(0):\n')))
func_value.append(int(input('input f(1):\n')))
func_value.append(int(input('input f(2):\n')))
func_value.append(int(input('input f(3):\n')))
print('f(0)=%d' %(func_value[0]))
print('f(1)=%d' %(func_value[1]))
print('f(2)=%d' %(func_value[2]))
print('f(3)=%d' %(func_value[3]))
print('Programming the circuit...')

init(QMachineType.CPU_SINGLE_THREAD)

qubit_num = 4
cbit_num = 2
# Initialization of 4 quantum bits
qv = qAlloc_many(qubit_num)
cv = cAlloc_many(cbit_num)
simonAlgorithm = Simon_QProg(qv, cv, func_value)

result = []
for i in range(0, 20, 1):
    re = directly_run(simonAlgorithm)
    result.append(cv[0].eval()*2 + cv[1].eval())
if 3 in result:
    if 2 in result:
        print('s=00')
    else:
        print('s=11')
elif 2 in result:
    print('s=01')
elif 1 in result:
    print('s=10')

finalize()

```

C++

```

@settings:
    language = C++;
    autoimport = True;

```

(continues on next page)

(continued from previous page)

```

    compile_only = False;

@qcodes:
circuit controlfunc(vector<qubit> qvec, int index, int value) {
    let cd = qvec.length() / 2;
    vector<qubit> qvtemp;
    qvtemp = qvec[0:cd];
    if (index == 1) {
        X(qvec[0]);
    } else if (index == 2) {
        X(qvec[1]);
    } else if (index == 0) {
        X(qvec[0]);
        X(qvec[1]);
    }

    if (value == 1) {
        X(qvec[3]).control(qvtemp);
    } else if (value == 2) {
        X(qvec[2]).control(qvtemp);
    } else if (value == 3) {
        X(qvec[2]).control(qvtemp);
        X(qvec[3]).control(qvtemp);
    }

    if (index == 1) {
        X(qvec[0]);
    } else if (index == 2) {
        X(qvec[1]);
    } else if (index == 0) {
        X(qvec[0]);
        X(qvec[1]);
    }
}

circuit oraclefunc(vector<qubit> qvec, vector<int> funvalue) {
    let cd = qvec.length()/2;
    for (let i=0: 1: 4){
        let value = funvalue[i];
        controlfunc(qvec, i, value);
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
}

Simon_QProg(vector<qubit> qvec, vector<cbit> cvec, vector<int> funvalue) {
    let cd = cvec.length();
    for (let i=0: 1: cd) {
        H(qvec[i]);
    }
    oraclefunc(qvec, funvalue);
    for (let i=0: 1: cd) {
        H(qvec[i]);
    }
    for (let i=0: 1: cd) {
        measure(qvec[i], cvec[i]);
    }
}

@script:
int main() {
    cout << "4-qubit Simon Algorithm\n" << endl;
    cout << "f(x)=f(y)\t x+y=s" << endl;
    cout << "input f(x),f(x):[0,3]" << endl;
    vector<int> funcvalue(4, 0);
    cout << "input f(0):" << endl;
    cin >> funcvalue[0];
    cout << "input f(1):" << endl;
    cin >> funcvalue[1];
    cout << "input f(2):" << endl;
    cin >> funcvalue[2];
    cout << "input f(3):" << endl;
    cin >> funcvalue[3];
    cout << "f(0)=" << funcvalue[0] << endl;
    cout << "f(1)=" << funcvalue[1] << endl;
    cout << "f(2)=" << funcvalue[2] << endl;
    cout << "f(3)=" << funcvalue[3] << endl;
    cout << " Programming the circuit..." << endl;
    init(QMachineType::CPU);
    int qubit_num = 4;
    int cbit_nun = 2;
    QVec qVec = qAllocMany(4);
    vector<ClassicalCondition> cVec = cAllocMany(2);

```

(continues on next page)

(continued from previous page)

```

QProg  simonAlgorithm = Simon_QProg(qVec, cVec, funcvalue);
vector<int> result(20);

for (auto i = 0; i < 20; i++) {
    directlyRun(simonAlgorithm);
    result[i] = cVec[0].eval() * 2 + cVec[1].eval();
}
if (find(result.begin(), result.end(), 3) != result.end()) {
    if (find(result.begin(), result.end(), 2) != result.end()) {
        cout << "s=00" << endl;
    } else {
        cout << "s=11" << endl;
    }
}
else if (find(result.begin(), result.end(), 2) != result.end()) {
    cout << "s=01" << endl;
}
else if (find(result.begin(), result.end(), 1) != result.end()) {
    cout << "s=10" << endl;
}
finalize();
}

```

### 7.6.3 6.6.3 Simon 算法小结

在一台量子计算机上运行了该算法的最简单版本，仅仅用了六个量子比特，量子计算机完成这一任务仅用了两次迭代，而普通计算机得用三次。这种区别似乎不算什么，但人们相信，如果增加更多量子比特，量子计算机和普通计算机运算能力的差别就会拉大，这也意味着，量子计算机能更快、更高效地解决此类算法问题。不过，还是要泼一盆冷水，到目前为止，能够运行西蒙算法并没有什么实际价值，该实验的唯一目的是证明量子计算机在一种算法上能够做得更好。

## 7.7 6.7 CoinFlip 算法

### 7.7.1 6.7.1 CoinFlip 算法介绍

**问题描述：**

E.D.Schell 在 1945 年 1 月版的美国数学月刊上提出了假币问题，初始问题是给定一定数量的硬币，其中一枚假币的质量和真币的质量不一样（外形一样，但质量轻），给定一个天平，可用来确定哪一枚是假币（通过天平的倾斜与否来判断重量差异）。

**理解问题：（以下是帮助理解规则）**

假设给定 10 枚硬币，其中一枚硬币是假的，质量比真硬币轻。

One of these coins is fake



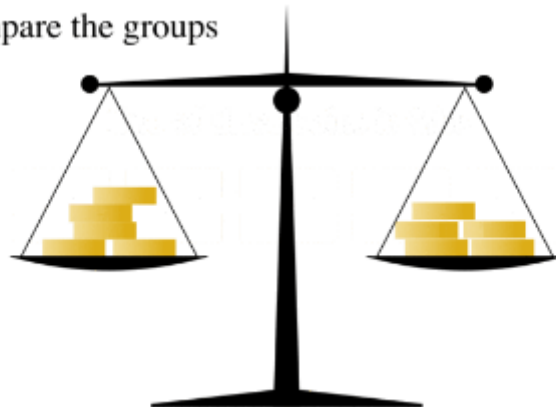
第一步，将 10 枚硬币均分为两组，每一组包含 5 个硬币：

Split the coins into two groups

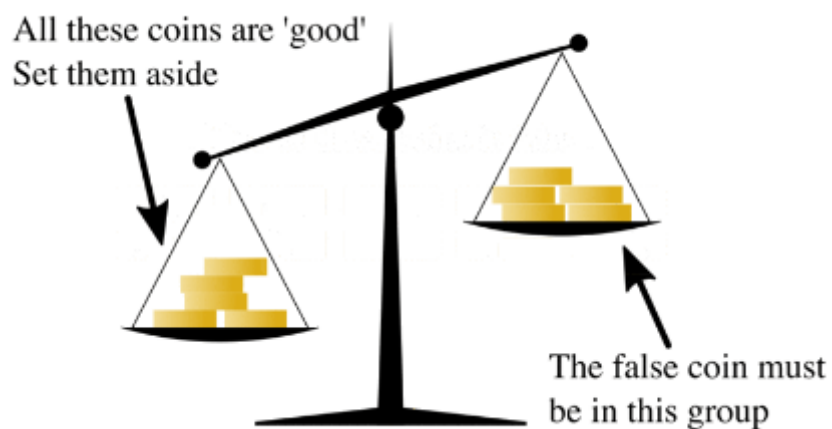


第二步，使用天平来判断：

Compare the groups



第三步：根据倾斜程度，可以判断假币在哪一个分组里，如下图，假硬币在左侧。



第四步，将包含假币的 5 枚硬币拿出来。

One of the remaining coins is fake



第五步，将 5 枚硬币划分为 2 组 2 枚的，外加单独 1 枚。

Split the remaining coins into two groups

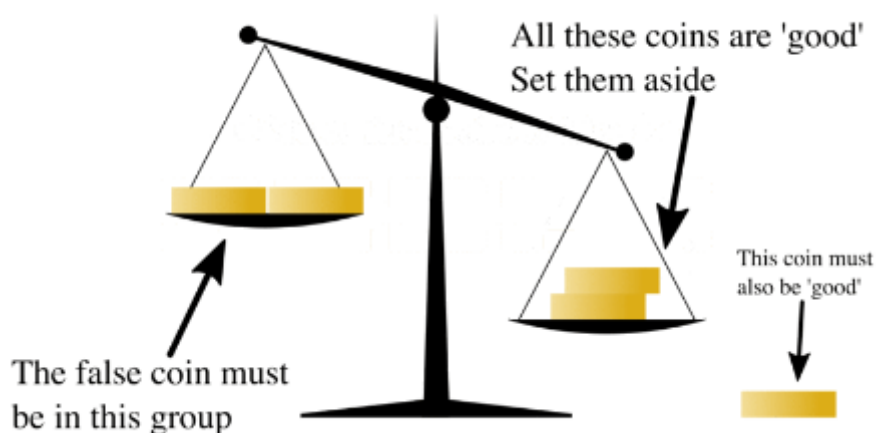


Set one coin aside since  
there are an odd number  
of coins

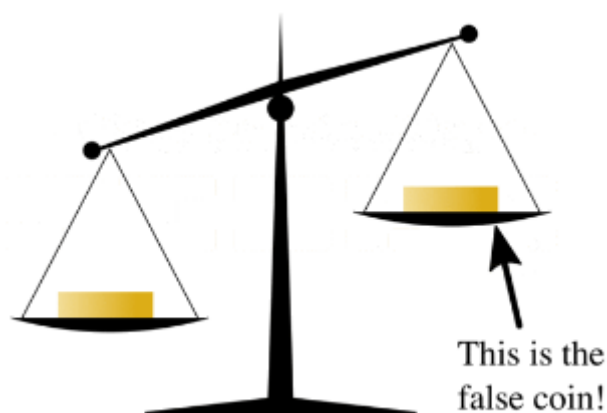
第六步，把 2 组硬币放入天平，查看天平倾斜情况。如果天平，持平，则额外的哪一枚是假币。



第七步，判断天平倾斜情况，如果是下面的情况，表明左侧包含了假币。



第八步，判断可能包含假币的两枚，分别放入天平两侧，一次性就可以判断出真假。



如上问题是对十个硬币的判断。（如上参考维基百科）

当然，该问题在不同的参考文献里有不同的版本，本实验算法里假设：

1. 真币的重量均等，假币的质量也均等，假币的质量比真币轻。
2. 天平只给我们提供两个信息，平衡（两组币的重量相同）或倾斜。

算法简述：给定  $N$  个外形一样的硬币，其中有  $k$  个假币，真币的质量均相等，假币质量轻。

算法目标：找出这  $k$  个假币。

### 策略：

本实验算法主要是 Bernstein 和 Vazirani 奇偶校验问题的一个应用，在经典策略里面，每次测量只能有一次，左右两边相同数的硬币数判断。而量子算法是通过构建叠加态从而对经典策略基础上的改进，我们可以同时查询叠加的左右两边状态。

## 量子策略模型简述：

在该问题里，平衡的天平模型可以用一个 Oracle 来刻画。简称 B-Oracle(Balance Oracle)，它是一个  $N$  位的寄存器： $x_1x_2...x_N \{0,1\}^N$ ，为了检索这些值，我们需要做一个查询 (Query)，查询字符  $q_1q_2...q_n \{0,1,-1\}^N$ ，其中包括相同数量的 1 和 -1 (数量定义为  $L$ )。该 Oracle 返回 1 位的答案  $X$ ，定义如下：

$$f(x) = \begin{cases} 0 & \sum_{i=1}^n x_i q_i = 0 \\ 1 & \text{Otherwise} \end{cases}$$

考虑  $x_1x_2...x_N$  表示  $N$  个硬币，而且 0 表示硬币质量均等，1 表示有一个假币。因此， $q_i = 1$  意味着我们把硬币  $x_i$  放在天平右侧， $q_i = -1$  则表示将  $x_i$  放在左侧盘里。这个时候我们必须保证，有相同数量的 1 和 -1 (天平里左右两侧放入相同数量的硬币)，答案  $X$  正确的模拟了天平秤。如： $X = 0$  则表示天平两边相等，反之  $X = 1$  表示倾斜。

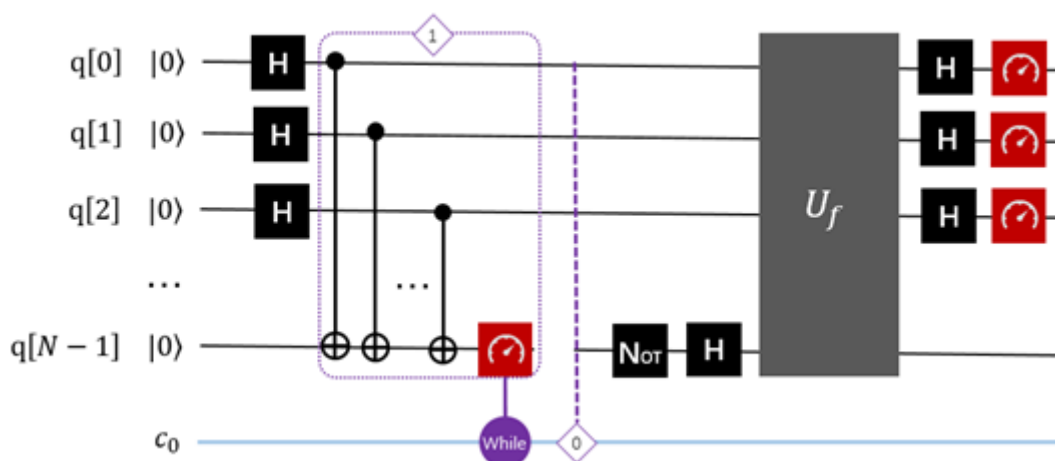
有效的构造转化  $W$ (这里的思想可以参考 Gervor 算法)。从上面我们也可看出，如果  $N$  个硬币里包含了  $k$  ( $k$  大于 1) 个假币，那么  $\text{Find}(k)$  的复杂度是多项式时间复杂，我们已经在 Bernstein-Vazirani 算法接触了  $k=2$  的情况 (请参考我们算法库里的 Bernstein-Vazirani Algorithm)。本算法主要目的是展示量子算法的优越性，因此只考虑包含一个假币的情况 (即  $k=1$ )。

查找所有  $k$  假硬币的量子查询复杂度是入下表，给定输入如上描述。

| Results   | k=1      | k=2               | k=3               | general               |
|-----------|----------|-------------------|-------------------|-----------------------|
| Quantum   | 1        | 1                 | $2 \leq k \leq 3$ | $O(k^{1/4})$          |
| Classical | $\log N$ | $\geq 2\log(N/2)$ | $\geq 3\log(N/3)$ | $\Omega(k \log(N/k))$ |

通过上表，比较清晰的展示了量子策略的优越性，尤其在多假币的情况下，当然，我们一个假币的情况，但是在硬币为  $N$  的时候，量子测量一次就可以完成。一个假币的情况，详情请看 Counterfeit Coin Game 的参考线路图：

## 参考线路图：





线路说明：紫色的 if 表示的是测量判断，根据输出的经典信息来判断是否需要执行下一步的操作。上面线路图里判定条件，如果输出为 0 的时候，则需要执行 0 对应的操作，实际上就是从新执行一遍量子线路，反之，执行 U<sub>f</sub> 操作，U<sub>f</sub> 指代了错误币所在位置的控制非门，目标位最后一位。

## 7.7.2 6.7.2 CoinFlip 算法的实现

下面给出 QRunes 实现 CoinFlip 算法的代码示例：

Python

```
@settings:
    language = Python;
    autoimport = True;
    compile_only = False;

@qcodes:
// Determine whether the next step needs to be performed based on the classical output
↪information
CoinFlip_Algorithm(vector<qubit> qlist, vector<cbit> clist, bool fx) {
    X(qlist[0]);
    H(qlist[0]);
    X(qlist[1]);
    CNOT(qlist[0], qlist[1]);
    H(qlist[1]);
    // If the output is 0, then the corresponding operation is needed later.

    if (fx) {
        X(qlist[0]);
    }
    H(qlist[0]);
    CNOT(qlist[0], qlist[1]);
    H(qlist[0]);
    measure(qlist[0], clist[0]);
    measure(qlist[1], clist[1]);
}

@script:
import sys
def CoinFlip_Prog(prog, q, c, fx):
    temp = CoinFlip_Algorithm(q, c, fx)
    prog.insert(temp)
    res = directly_run(prog)
```

(continues on next page)

(continued from previous page)

```

        return ( c[1].eval() << 1) + int(c[0].eval())

if __name__ == '__main__':
    print('Entanglement Flip Game')
    fx = int(input('Input choice of Q:(0/1)\n'))
    print('Programming the circuit...')

    init(QMachineType.CPU_SINGLE_THREAD)

    qubit_num = 2
    cbit_num = 2
    # Initialization of 2 quantum bits
    qv = qAlloc_many(qubit_num)
    cv = cAlloc_many(cbit_num)
    out_come = 0
    prog = QProg()
    temp = CoinFlip_Prog(prog, qv, cv, fx)
    for i in range(0, 10, 1):
        out_come = CoinFlip_Prog(prog, qv, cv, fx)
        if out_come != temp:
            print('Q wins!')
            sys.exit(0)
    print('max entanglement!')
    print('P wins!')

    finalize()

```

C++

```

@settings:
    language = C++;
    autoimport = True;
    compile_only = False;

@qcodes:
// Determine whether the next step needs to be performed based on the classical output
↪information
CoinFlip_Algorithm(vector<qubit> qlist, vector<cbit> clist, bool fx) {
    X(qlist[0]);
    H(qlist[0]);

```

(continues on next page)

(continued from previous page)

```

        X(qlist[1]);
        CNOT(qlist[0], qlist[1]);
        H(qlist[1]);
        // If the output is 0, then the corresponding operation is needed later.
        if (fx) {
            X(qlist[0]);
        }
        H(qlist[0]);
        CNOT(qlist[0], qlist[1]);
        H(qlist[0]);
        measure(qlist[0], clist[0]);
        measure(qlist[1], clist[1]);
    }

@script:
int CoinFlip_Prog(QProg & prog, vector<Qubit*> qVec, vector<ClassicalCondition> cVec,
    ↪ bool fx) {
    auto temp = CoinFlip_Algorithm(qVec, cVec, fx);
    prog << temp;
    directlyRun(prog);
    return ((1 << cVec[1].eval()) + (int)cVec[0].eval());
}

int main() {
    bool fx = 0;
    cout << "Entanglement Flip Game\n" << endl
         << "\n" << endl
         << "Input choice of Q:(0/1)\n";

    cin >> fx;
    cout << "Programming the circuit..." << endl;
    int outcome = 0;
    init(QMachineType::CPU);
    vector<Qubit*> qVec = qAllocMany(2);
    vector<ClassicalCondition> cVec = cAllocMany(2);
    QProg prog;
    auto temp = CoinFlip_Prog(prog, qVec, cVec, fx);
    for (auto i = 0; i < 10; i++) {
        outcome = CoinFlip_Prog(prog, qVec, cVec, fx);
        if (temp != outcome) {
            cout << "Q wins!\n" << endl;

```

(continues on next page)

(continued from previous page)

```

        return 0;
    }

    }
    cout << "max entanglement!" << endl;
    cout << "P wins!\n" << endl;
}

```

### 7.7.3 6.7.3 CoinFlip 算法小结

我们传统的电脑构建模块，只能存储两个状态中的其中一个，就如硬币，50 个同时抛掷你只能记录一种正反面的状态，50 个硬币同时记录的话，就需要量子计算机就数千兆字节的数据存储才能达到。量子计算机就是这样的，它们是基于量子位的，它可以同时处于两个状态。这可以使每个硬币的单个量子位一次存储所有配置的概率分布

## 7.8 6.8 Bernstein-Vazirani 算法

### 7.8.1 6.8.1 Bernstein-Vazirani 算法介绍

量子计算机是相对经典计算机而言的，量子计算机并不是在通常的计算问题上取代传统的电子计算机，而是针对特定问题完成经典计算机难以胜任的高难度计算工作。它是以量子力学为基础，实现量子计算的机器。比如：若运 Deutsch-Jozsa 问题的量子算法（DJ 算法），只需运行一次，就可以分辨函数是常数函数还是对称函数，而运用相应的经典算法则需要运行  $O(N)$  次才能达到该目的。后来，Bernstein 和 Vazirani 运用 DJ 算法有效地解决了询问量子数据库的问题（即 BV 算法）。

#### 问题描述：

Input:

考虑一个经典的布尔函数：

$$f : \{0,1\}^n \rightarrow \{0,1\}$$

存在  $s \in \{0,1\}^n$ ，再定义一个函数：

$$f_s(x) = \langle s, x \rangle, x \in \{0,1\}^n$$

$s$  是一个未知的向量，通常称  $s$  为隐藏字符串（Hidden string），其中  $\langle s, x \rangle$  表示内积（inner product），定义为：

$$\langle s, x \rangle = s_0 x_0 \oplus s_1 x_1 \oplus \dots \oplus s_n x_n$$

符号  $\oplus$  在所出现的量子算法文中都表示布尔加或模 2 加。)

Output:

算法目标：找到  $s$ 。

### 经典算法情况：

由于对该函数的每个经典查询只能生成 1 位的信息，而任意隐藏字符串  $s$  具有  $n$  位的信息，所以经典查询复杂度是  $O(n)$ 。

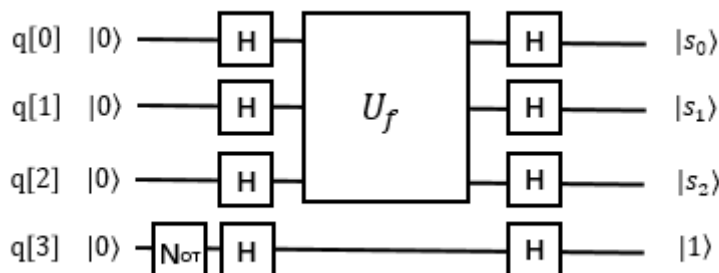
### 量子算法情况：

Bernstein-Vazirani 的工作建立在 Deutsch 和 Jozsa 早期工作理论上探索量子查询复杂度。他们对该领域的贡献是一个用于隐藏字符串问题的量子算法，该算法的非递归量子查询复杂度仅为 1，同比经典情况  $O(n)$ 。这一量子算法的真正突破在于加快查询复杂度，而不是执行时间本身。

案例：考虑  $n=3$  时的 Bernstein-Vazirani 问题。变量是 3 比特时，二进制表示为  $x_0 x_1 x_2$ ，常数  $s$  则表示为  $s_0 s_1 s_2$ ，因此所求的常数  $s$  总共有 8 个。此时，问题函数描述如下：

$$f_s(x_0 x_1 x_2) = s_0 x_0 \oplus s_1 x_1 \oplus s_2 x_2$$

不难看出，对于经典算法而言，如果是  $f_s(100) = s_0$ ， $f_s(010) = s_1$ ， $f_s(001) = s_2$ ，那么最少也需要 3 次调用函数才可以确定常量  $s = s_0 s_1 s_2$ 。但是对于量子算法而言，使用下面的量子 Oracle 计算，1 次就可以决定  $s = s_0 s_1 s_2$ ，其计算复杂度为  $O(1)$ 。



分析上图：

$$\begin{aligned}
 |000\rangle|1\rangle &\xrightarrow{H\ H\ H\ H} \frac{2}{2\sqrt{2}} \sum_{x=0}^7 |x\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \\
 &\xrightarrow{U_f} \frac{1}{2\sqrt{2}} \sum_{x=0}^7 (-1)^{\langle s, x \rangle} |x\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \\
 &\xrightarrow{H\ H\ H\ H} \frac{2}{2\sqrt{2}} \sum_{x=0, y=0}^7 (-1)^{\langle s, x \rangle \langle x, y \rangle} |y\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \equiv |s\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right)
 \end{aligned}$$

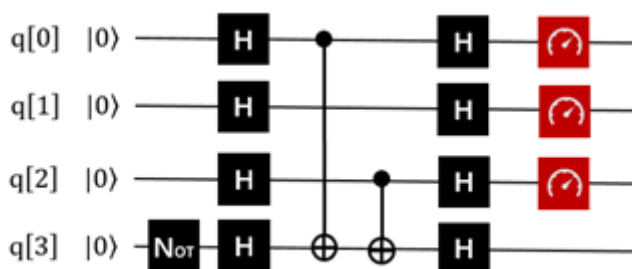
不失一般性：

$$\begin{aligned}
 |0^n\rangle|1\rangle &\xrightarrow{H^{(n+1)}} \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \\
 &\xrightarrow{U_f} \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (-1)^{\langle s, x \rangle} |x\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \\
 &\xrightarrow{H^{(n+1)}} \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (-1)^{\langle s, x \rangle \langle x, y \rangle} |y\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \equiv |s\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right)
 \end{aligned}$$

参考线路图：

下面给出两组案例，分别是  $s=101$  和  $s=111$

$s=101$



QRunes :

```

RX 3,"pi"
H 0
H 1
H 2
H 3
CNOT 0,3
    
```

(continues on next page)

(continued from previous page)

```

CNOT 2,3
H 0
H 1
H 2
MEASURE 0,$0
MEASURE 1,$1
MEASURE 2,$2

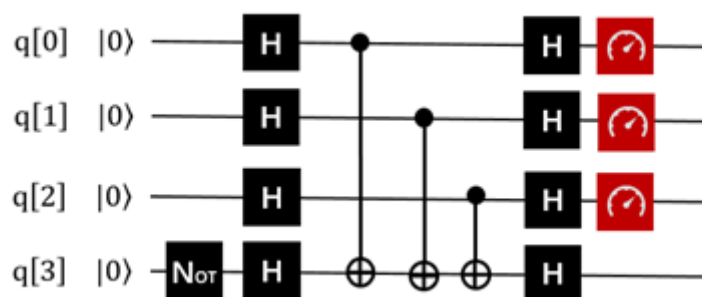
```

这时，输出的结果，指代了  $s$ 。通过验证，输出结果为：



$s=111$  时：

线路图设计为：



测量结果：



QRunes :

```
RX 3,"pi"  
H 0  
H 1  
H 2  
H 3  
CNOT 0,3  
CNOT 1,3  
CNOT 2,3  
H 0  
H 1  
H 2  
MEASURE 0,$0  
MEASURE 1,$1  
MEASURE 2,$2
```

## 7.8.2 6.8.2 Bernstein-Vazirani 算法的实现

下面给出 QRunes 实现 Bernstein-Vazirani 算法的代码示例:

Python

```
@settings:  
    language = Python;  
    autoimport = True;  
    compile_only = False;  
  
@qcodes:  
circuit<vector<qubit>,qubit> generate_bv_oracle(vector<bool> oracle_function){  
    return lambda (vector<qubit> qVec, qubit qu): {  
        let cd = oracle_function.length();  
  
        for(let i = 0: 1: cd){  
            if(oracle_function[i]){  
                CNOT(qVec[i], qu);  
            }  
        }  
    }  
};
```

(continues on next page)



(continued from previous page)

```

}

BV_QProg(vector<qubit> qVec, vector<cbit> cVec, vector<bool> a, circuit<vector<qubit>,
↪qubit> oracle){

    if(qVec.length() != (a.length()+1)){
        let cd = qVec.length();
        X(qVec[cd-1]);
        apply_QGate(qVec, H);
        oracle(qVec, qVec[cd - 1]);

        qVec.remove(0);

        apply_QGate(qVec, H);
        measure_all(qVec, cVec);
    }
}

}

@script:
import sys
if __name__ == '__main__':
    print('Bernstein Vazirani Algorithm')
    print('f(x)=a*x+b')
    input_a = input('input a\n')
    a = []
    for i in input_a:
        if i == '0':
            a.append(0)
        else:
            a.append(1)
    b = int(input('input b\n'))
    print('a=\t%s' %(int(input_a)))
    print('b=\t%s' %(int(bool(b))))
    print('Programming the circuit...')

    init(QMachineType.CPU_SINGLE_THREAD)
    qubit_num = len(a)
    cbit_num = qubit_num
    # Initialization quantum bits
    qv = qAlloc_many(qubit_num+1)
    cv = cAlloc_many(cbit_num)

```

(continues on next page)

(continued from previous page)

```

if len(qv) != (len(a)+1):
    print("error: param error")
    sys.exit(1)
bvAlgorithm = BV_QProg(qv, cv, a, b)
directly_run(bvAlgorithm)

print('a=\t', end='')
for c in cv:
    print(c.eval())
print('b=\t%s' %(int(bool(b))))

finalize()

```

C++

```

@settings:
    language = C++;
    autoimport = True;
    compile_only = False;

@qcodes:
circuit<vector<qubit>,qubit> generate_bv_oracle(vector<bool> oracle_function){
    return lambda (vector<qubit> qVec, qubit qu): {
        let cd = oracle_function.length();

        for(let i = 0: 1: cd){
            if(oracle_function[i]){
                CNOT(qVec[i], qu);
            }
        }
    };
}

BV_QProg(vector<qubit> qVec, vector<cbit> cVec, vector<bool> a, circuit<vector<qubit>,
↪qubit> oracle){

    if(qVec.length() != (a.length()+1)){
        let cd = qVec.length();
        X(qVec[cd-1]);
    }
}

```

(continues on next page)

(continued from previous page)

```

        apply_QGate(qVec, H);
        oracle(qVec, qVec[cd - 1]);

        qVec.remove(0);

        apply_QGate(qVec, H);
        measure_all(qVec, cVec);
    }

}

@script:
int main() {
    cout << "Bernstein Vazirani Algorithm\n" << endl;
    cout << "f(x)=a*x+b\n" << endl;
    cout << "input a" << endl;
    string stra;
    cin >> stra;
    vector<bool> a;
    for (auto iter = stra.begin(); iter != stra.end(); iter++)
    {
        if (*iter == '0')
        {
            a.push_back(0);
        }
        else
        {
            a.push_back(1);
        }
    }
    cout << "input b" << endl;
    bool b;
    cin >> b;
    cout << "a=\t" << stra << endl;
    cout << "b=\t" << b << endl;
    cout << " Programming the circuit..." << endl;
    size_t qubitnum = a.size();
    init();

```

(continues on next page)

(continued from previous page)

```

vector<Qubit*> qVec = qAllocMany(qubitnum+1) ;
auto cVec = cAllocMany(qubitnum);
auto oracle = generate_bv_oracle(a);
auto bvAlgorithm = BV_QProg(qVec, cVec, a, oracle);
directlyRun(bvAlgorithm);
string measure;
cout << "a=\t";
for (auto iter = cVec.begin(); iter != cVec.end(); iter++)
{
    cout << (*iter).eval();
}
cout << "\n" << "b=\t" << b << endl;
finalize();
}

```

### 7.8.3 6.8.3 Bernstein-Vazirani 算法小结

Bernstein-Vazirani 的工作建立在 Deutsch 和 Jozsa 早期工作理论上探索量子查询复杂度。他们对该领域的贡献是一个用于隐藏字符串问题的量子算法, 该算法的非递归量子查询复杂度仅为 1, 同比经典情况  $O(n)$ 。这一量子算法的真正突破在于加快查询复杂度, 而不是执行时间本身。

## 7.9 6.9 QPE 算法

### 7.9.1 6.9.1 QPE 算法介绍

QPE 算法 (Quantum Phase Estimation), 量子相位估计算法。该算法在很早就已经被提出了, 然而真正带来很大影响的, 就是基于它实现的 HHL 算法, 以及各种基于 HHL 算法实现的量子机器学习算法。相位估计最神奇的效果是达到了相比传统计算机上运行的算法的一个指数加速。这个效果怎么理解, 举个简单的例子, 就是传统计算机上要运行 2 的 30 次方 (约等于 10 亿) 次运算, 量子计算机上只要运行 30 次左右就 OK 了。当然这个指数加速效果是有前提条件的, 就是输入和输出都需要是量子比特, 而量子比特怎么和经典比特对应起来呢, 这又是另外一个问题了。我们只要知道, 在算法的运算过程中, 是有一个指数加速的效果就可以了。而这个算法凭借它的优势, 可以用在很多应用上, 比如求阶问题, 因子分解问题以及量子机器学习 HHL 算法等等。

#### 问题描述:

相位估计 (Phase estimation), 也简称 QPE, 是量子计算中极为重要的一个子算法, 它作为很多算法的主要构成部分, 包括 Shor 算法, HHL 求解线性系统算法等。尤其 HHL 算法, 因为求解线性系统对量子机器学习

习算法的的价值潜力极大。目前，可用的真实量子比特可用数还比较少，如果我们想要测量更复杂的可观测量，例如哈密顿量  $H$  描述的能量，我们会采用量子相位估计。

在相位估计里会用到量子傅里叶变换（QFT），它的重要性体现在它是很多量子算法的基础，因此了解本算法，对于了解其他量子算法有很好的帮助，甚至是理解很多典型算法必不可少的路径。

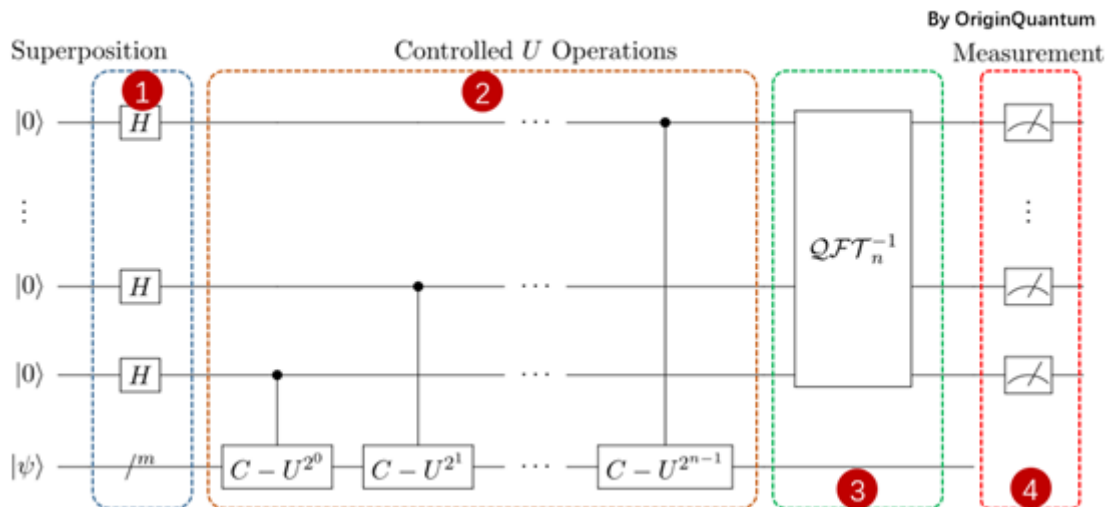
### 问题定义：

给定可作用于量子线路的幺正矩阵  $U$  以及其本征向量之一  $|\psi\rangle$ ，求其对应的本征值。

$$U|\psi\rangle = e^{2\pi i \phi} |\psi\rangle$$

由于  $U$  是幺正矩阵，所以其本征值可以被表示为  $e^{2\pi i \phi}$ 。求本征值在这里等价于求相位  $\phi$ ，相位估计，顾名思义，就是求解相位  $\phi$ 。该算法使用受控  $U$  操作  $O(1/\epsilon)$  在加性误差  $\epsilon$  内，可以高概率估计  $\phi$  的值。

需要准备实现的线路模型图如下：



### 植入步骤：

- 初始化：

输入包括两部分寄存器集：上面的  $n$  个量子位包含第一个寄存器，下面的  $m$  个量子位是第二个寄存器。

- 创建叠加态：

统一给  $n$  个量子比特添加 H 门操作，将  $n$  个初始状态置于叠加态，上图中淡蓝色框步骤 1 里。

- 应用受控的单一操作：

C-U 是一个受控 U 门，只有当相应的控制位（来自第一个寄存器）为  $|1\rangle$  时，它才在第二个寄存器上应用单一运算符 U。注意位置位的位置，与目标位的对应。

- 应用逆傅里叶变换：

执行傅里叶变换，详情参考傅里叶变换算法。

- 测量：

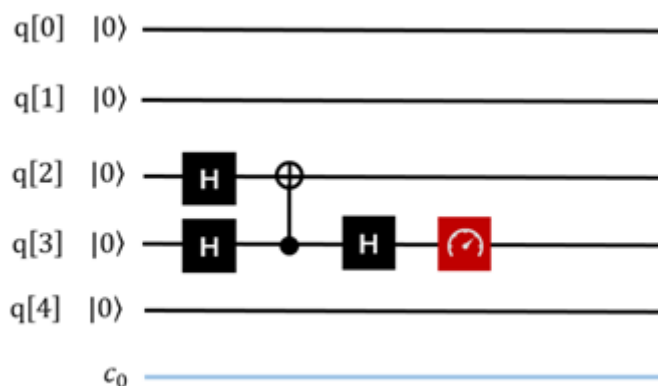
测量前量子比特。

如上就是相位估计算法的步骤。需要需要注意的是受控单元那里的处理。

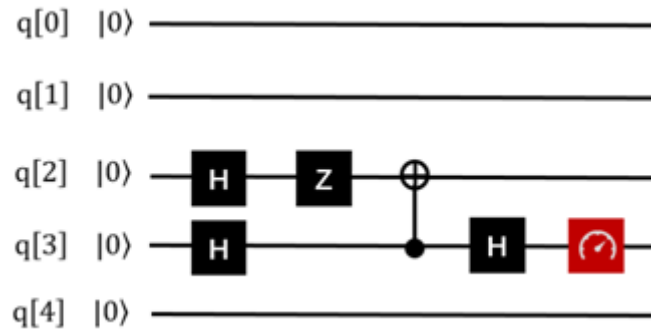
### 参考线路图：

这里给出了一个最简单的测试案例，分别是“-”的相位估计和“+”的相位估计。以下示例演示了作用于量子比特  $q[2]$  的单量子比特哈密顿量  $x$  的量子相位估计。去  $q[3]$  作为指针系统（测量操作在上面）。在这个例子中，指针系统上的量子傅立叶变换等效于  $q[3]$  上的 Hadamard 门。CNOT 门描述了系统 + 指针系统的离散化演变。指针量子位  $q[3]$  的最终测量结果是 0 或 1，这取决于  $q[2]$  是在  $x$  的  $+1$  还是  $-1$  对应的本征态中准备。在该示例中，量子位  $q[2]$  初始化为  $ZH|0\rangle$ ，它是  $x = -1$  所对应的特征向量。因此，测量结果为 1。（注意，测量结果等于 0，表示的就是  $-1$ ）

### 相位估计 - 的情况：



相位估计 + 情况：



### 7.9.2 6.9.2 QPE 算法的实现

下面给出 QRunes 实现 QPE 算法的代码示例：

Python

```
@settings:
    language = Python;
    autoimport = True;
    compile_only = False;

@qcodes:
// Perform Fourier Transform
circuit QFT(vector<qubit> q) {
    for(let i=0: 1: q.length()) {
        H(q[q.length()-1-i]);
        for (let j=i+1: 1: q.length()) {
            CR(q[q.length()-1-j],
                q[q.length()-1-i], 2*Pi/(1<<(j-i+1)));
        }
    }
}

circuit QFTdagger(vector<qubit> q) {
    QFT(q).dagger();
}

circuit unitary(vector<qubit> q) {
    RX(q[0], -Pi);
}
```

(continues on next page)

(continued from previous page)

```

}

circuit Hadamard(vector<qubit> q) {
    for (let i=0: 1: q.length()) {
        H(q[i]);
    }
}

// U is generated by qc,
// output  $U^{(2^{\min})}$ 
circuit unitarypower(vector<qubit> q, int min) {
    for (let i=0: 1: (1 << min)) {
        unitary(q);
    }
}

// Applying a controlled single operation
circuit controlUnitaryPower(vector<qubit> qvec, qubit controlQubit, int min) {

    unitarypower(qvec, min).control(controlQubit);
}

circuit QPE(vector<qubit> controlqvec, vector<qubit> targetqvec) {
    // A unified H-gate operation is added to controlqvec qubits, and N initial states
    // are placed in superposition states.
    for(let i=0: 1: controlqvec.length()) {
        H(controlqvec[i]);
    }

    vector<qubit> controlqubit;

    for (let i=0: 1: controlqvec.length()) {
        controlUnitaryPower(targetqvec, controlqvec[controlqvec.length()-1-i], i);
    }
    QFTdagger(controlqvec);
}

@script:
if __name__ == '__main__':
    init(QMachineType.CPU_SINGLE_THREAD)

```

(continues on next page)



(continued from previous page)

```

qubit_num = 2
cbit_num = 2
# Initialization 2 quantum bits
cq = qAlloc_many(qubit_num)
# Initialization 1 quantum bits
tq = qAlloc_many(1)
cv = cAlloc_many(cbit_num)
qpeProg = QProg()
qpeProg.insert(H(tq[0]))
qpeProg.insert(QPE(cq, tq))
qpeProg.insert(measure(cq[0], cv[0]))
qpeProg.insert(measure(cq[1], cv[1]))
directly_run(qpeProg)
print("c0: %s" %(cv[0].eval()))
print("c1: %s" %(cv[1].eval()))

finalize()

```

C++

```

@settings:
    language = C++;
    autoimport = True;
    compile_only = False;

@qcodes:
// Perform Fourier Transform
circuit QFT(vector<qubit> q) {
    for(let i=0: 1: q.length()) {
        H(q[q.length()-1-i]);
        for (let j=i+1: 1: q.length()) {
            CR(q[q.length()-1-j],
                q[q.length()-1-i], 2*Pi/(1<<(j-i+1)));
        }
    }
}

circuit QFTdagger(vector<qubit> q) {
    QFT(q).dagger();
}

```

(continues on next page)

(continued from previous page)

```

}

circuit unitary(vector<qubit> q) {
    RX(q[0], -Pi);
}

circuit Hadamard(vector<qubit> q) {
    for (let i=0: 1: q.length()) {
        H(q[i]);
    }
}

// U is generated by qc,
// output  $U^{(2^{\min})}$ 
circuit unitarypower(vector<qubit> q, int min) {
    for (let i=0: 1: (1 << min)) {
        unitary(q);
    }
}

// Applying a controlled single operation
circuit controlUnitaryPower(vector<qubit> qvec, qubit controlQubit, int min) {

    unitarypower(qvec, min).control(controlQubit);
}

circuit QPE(vector<qubit> controlqvec, vector<qubit> targetqvec) {
    // A unified H-gate operation is added to controlqvec qubits, and N initial states
    // are placed in superposition states.
    for(let i=0: 1: controlqvec.length()) {
        H(controlqvec[i]);
    }

    vector<qubit> controlqubit;

    for (let i=0: 1: controlqvec.length()) {
        controlUnitaryPower(targetqvec, controlqvec[controlqvec.length()-1-i], i);
    }

    QFTdagger(controlqvec);
}

```

(continues on next page)

(continued from previous page)

```

}

@script:
int main()
{
    init(QMachineType::CPU);
    int qubit_number = 2;
    int cbitnum = 2;
    vector<Qubit*> cqv = qAllocMany(qubit_number);          //control
    vector<Qubit*> tqv = qAllocMany(1);
    vector<ClassicalCondition> cv = cAllocMany(cbitnum);
    auto qpeProg = CreateEmptyQProg();
    qpeProg << H(tqv[0]);
    qpeProg << QPE(cqv, tqv);
    qpeProg << Measure(cqv[0], cv[0]) << Measure(cqv[1], cv[1]);
    directlyRun(qpeProg);
    cout << "c0: " << cv[0].eval() << endl;
    cout << "c1: " << cv[1].eval() << endl;
    finalize();
}

```

### 7.9.3 6.9.3 QPE 算法小结

QPE 算法应用了量子傅里叶逆变换，同时作为一个实用的基础量子算法，又被应用在 Shor Algorithm（质因数分解算法），和 HHL Algorithm（经常用于各种量子机器学习的最优化算法）等等算法中。

## 7.10 6.10 Shor 算法

### 7.10.1 6.10.1 Shor 算法介绍

舒尔算法，即秀尔算法（Shor 算法），以数学家彼得·秀尔命名，是一个在 1994 年发现的，针对整数分解这题目的量子算法（在量子计算机上面运作的算法）。它解决如下题目：给定一个整数  $N$ ，找出他的质因数。

在一个量子计算机上面，要分解整数  $N$ ，秀尔算法的运作需要多项式时间（时间是  $\log N$  的某个多项式这么长， $\log N$  在这里的意义是输入的档案长度）。更精确的说，这个算法花费  $O((\log N)^3)$  的时间，展示出质因数分解问题可以使用量子计算机以多项式时间解出，因此在复杂度类 BQP 里面。这比起传统已知最快的因数分解算法，普通数域筛选法还要快了一个指数的差异。

秀尔算法非常重要，因为它代表使用量子计算机的话，我们可以用来破解已被广泛使用的公开密钥加密方法，

也就是 RSA 加密算法。RSA 算法的基础在于假设了我们不能很有效率的分解一个已知的整数。就目前所知，这假设对传统的（也就是非量子）电脑为真；没有已知传统的算法可以在多项式时间内解决这个问题。然而，秀尔算法展示了因数分解这问题在量子计算机上可以很有效率的解决，所以一个足够大的量子计算机可以破解 RSA。这对于建立量子计算机和研究新的量子计算机算法，是一个非常大的动力。

将两个质数乘起来，例如  $907 \times 641 = 581387$ ，是一件小学生都能做到的事情，用计算机去处理，看起来也没有什么难度。但是如果我给你 581387，让你去找它的质因数，问题就变得很复杂了。也许你可以用计算机一个一个的去尝试，但是当数字变得更大，达到成百上千位的时候，就连计算机也无能为力。世界上面有很多问题都是这样，难以找到答案，但是一旦找到答案就很容易去验证。类似的问题我们称之为 NP 问题。NP 问题之所以难于处理，是因为它的时间复杂度往往具有指数级别。这意味着随着问题规模的线性扩大，需要的时间却是指数增长的。利用这个原理，人们创造了 RSA 算法，它利用大数难以分解，但是易于验证的原理，对数据进行有效的加密。

量子计算机有将问题指数加速的能力，那是否意味着能攻克所有的 NP 问题呢？很遗憾，不能。但是幸运的是，我们有能力把“质因数分解”的时间复杂度降低到多项式级别，使大数分解问题的解决变为可能。这就是 Shor 算法。Shor 算法的提出意味着 RSA 密钥的安全性受到了挑战。下面我们就来介绍 Shor 算法的内容。

## 问题的转化

Shor 算法首先将质因数分解问题转换成了一个子问题，下面我们来看问题的转换过程。假设我们待分解的数为  $N$ ；

STEP 1: 随机取一个正整数  $1 < a < N$ ，定义一个函数:  $f(x) = 2^x \bmod N$ ；

STEP 2: 这个函数一定是一个周期函数，寻找到它的周期为  $r$ 。（这一步将使用量子计算机完成）；

STEP 3: 如果  $r$  为奇数，那么回到 STEP 1。如果  $r$  为偶数，那么计算  $f(r/2)$ ；

STEP 4: 如果  $f(r/2) = -1$ ，那么回到 STEP 1。否则，计算  $f(r/2) + 1$  和  $f(r/2) - 1$  分别对于  $N$  的最大公约数；

STEP 5: 这两个最大公约数就是  $N$  的两个质因数；

举个例子，对于 21 而言，假设我们选择  $a = 2$ ，那么

STEP 1: 定义函数  $f(x) = 2^x \bmod N$

STEP 2: 发现它的周期为 6。

STEP 3: 计算出  $f(3) = 8$

STEP 4: 计算 7 和 9 分别对于 21 的最大公因数  $\gcd(7, 21) = 7$   $\gcd(9, 21) = 3$

检验知 7 和 3 都是 21 的质因数，于是我们得到了问题的答案。

## 函数的引入

我们要为 STEP 1 中描述的函数找到它引入量子计算机的方式。这种函数被称为模指数 (Modular Exponential) 函数，在经典逻辑电路中，它已经被以各种形式设计了出来。所以现在，我们要为它准备一个量子线路的版本。

根据在“Oracle 是什么”这一节里面提到的量子函数概念，我们需要构建出一个酉变换  $U$  使得：

$$U|x|y\rangle = |x|y^{a^x} \bmod N\rangle$$

这种情况是一种比较普适的情况，我们令  $y = 1$ ，那么后面的这一组量子比特就作为辅助比特存储了  $f(x)$  的计算结果。我们先来找一种比较简单的情况来分析具体问题，可以便于对其中的变量分解转换的理解。选取要分解的质因数 15，和一个比 15 小的任意正整数 7，所以我们要构建这样的酉变换：

$$U|x|1\rangle = |x|7^x \bmod 15\rangle$$

首先要提到的一点是要表示  $7^x \bmod 15$ ，就意味着我们的辅助比特的取值是从 0~14 的，为了表示这个数，需要用到 4 个比特，即从 0000~1110。对于前面的工作比特来说，它的位数选择比较自由，而且选取的位数越多，我们得到正确结果的概率越大，这一点在后面会解释。

乍一看这个函数让我们有些无从下手，所以我们要对它进行一定的转换，比如先把  $x$  转化为二进制：

$$7^x = 7^{x_0 + 2x_1 + 2^2x_2 + \dots} = 7^{x_0} \cdot (7^2)^{x_1} \cdot (7^4)^{x_2} \dots \cdot (7^{2^n})^{x_n}$$

$x_i$  是  $x$  转换为二进制后每一位上对应的数码，所以它的取值无非是 0 或者 1。这样我们就可以简单的用一个控制酉操作得到每一项，即

$$\begin{aligned} |x_i\rangle = |1\rangle & : U_a|y\rangle \rightarrow |y \cdot 7^{2^i} \pmod{15}\rangle \\ |x_i\rangle = |0\rangle & : U_a = I \end{aligned}$$

其中  $I$  是单位操作。所以问题就转化为了构建“控制模乘”操作  $U_a$ 。

顺带一提，因为我们关注的点不是如何纯粹的用量子线路来描述里面的每一步操作，某些操作也不引入额外的计算时间复杂度，那么这些操作是可以用经典计算机代为完成的。比如说这里的  $7^{2^i}$ 。注意到

$$y \cdot 7^{2^i} \pmod{15} = (y \cdot (7^{2^{i-1}} \pmod{15})) \pmod{15}$$

我们只需要事先用经典计算机将  $7^{2^i} \pmod{15} (i = 0 \sim N-1)$  ( $N$  是选取的工作位数) 全部计算出来，就可以在接下来的设计时只考虑对应的几种情况。

我们可以看出， $a^{2^i} = a^{2^{i-1}+2^{i-1}} = (a^{2^{i-1}})^2$ ，根据这个公式，可以列举出来对于不同的  $i$  的取值情况，上述表达式的取值（这个过程用经典计算机就可以完成）。在例子中的这种情况中，有

$$\begin{aligned} i = 0 & \quad 7^{2^i} \pmod{15} = 7 \\ i = 1 & \quad 7^{2^i} \pmod{15} = 4 \\ i = 2 & \quad 7^{2^i} \pmod{15} = 1 \\ i \geq 3 & \quad 7^{2^i} \pmod{15} = 1 \end{aligned}$$

也就是说我们只需要对应设计  $U_a|y\rangle \rightarrow |7y \pmod{15}\rangle$ ,  $U_a|y\rangle \rightarrow |4y \pmod{15}\rangle$  两种就可以达到设计目的了。

最后我们来看一下引入了函数，量子态变成了什么。

首先是一组 Hadamard 变换，它们只作用在一组  $N$  个工作比特上，所以这个总状态就会变成

$$|\text{Working}\rangle|\text{Ancilla}\rangle = \left( \sum_{x=0}^{2^N-1} |x\rangle \right) |00\dots 001\rangle$$

在量子函数作用在这一组量子态时，相当于这个函数的自变量从 0 到  $2^N - 1$  的所有取值都被保存到了辅助比特上。也就是说，工作比特的每个状态分量都和辅助比特的一个状态分量纠缠在了一起。

$$\sum |x\rangle |f(x)\rangle$$

在之前的计算中，我们知道了  $f(x) = a^x \pmod{N}$  是一个周期函数，假设它的周期是  $T$ 。明显地，

$$f(x) = f(x+T) = f(x+2T)\dots$$

那么

$$|x\rangle |f(x)\rangle + |x+T\rangle |f(x+T)\rangle + |x+2T\rangle |f(x+2T)\rangle + \dots = (|x\rangle + |x+T\rangle + |x+2T\rangle + \dots) |f(x)\rangle$$

回到  $a = 7$ ,  $N = 15$  的例子中，我们有

$$\begin{aligned}
|\text{Working}\rangle|\text{Ancilla}\rangle = & (|0\rangle + |4\rangle + |8\rangle + \dots)|1\rangle \\
& + (|1\rangle + |5\rangle + |9\rangle + \dots)|7\rangle \\
& + (|2\rangle + |6\rangle + |10\rangle + \dots)|4\rangle \\
& + (|3\rangle + |7\rangle + |11\rangle + \dots)|13\rangle
\end{aligned}$$

因为这个态是一个纠缠态，所以当我们测量辅助比特时，工作比特就会坍缩成对应的那种情况。但是不论你得到辅助比特的测量值是什么，工作比特总是会只保留为每个分量都恰好为一组周期数的叠加态。那么这一组叠加态表示的数的周期将会通过量子傅里叶变换来快速完成。

### 量子傅里叶变换

寻找态的周期可以通过量子傅里叶变换来快速完成。我们先以  $|0\rangle + |4\rangle + |8\rangle + \dots$  为例子来看看量子傅里叶变换是怎么做的，之后你就会发现它对于 1,5,9,13... 或是 2,6,10,14... 都能得到类似的结果。

如图所示，量子傅里叶变换有两个重要的部分，第一是递归的依次控制旋转（CROT）操作，第二部分是改变比特的顺序。

数学表达上，每一项都是用离散傅里叶变换的形式去处理的。

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega^{jk}$$

其中  $x_j$  表示输入量子态的第  $j$  个分量，而  $k$  表示输出量子态的分量，如果用  $n$  个量子比特表示，则  $\omega = e^{\frac{2\pi i}{2^n}} = e^{\frac{2\pi i}{N}}$ 。而从矩阵上来看，则为

$$F_N = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

不妨假设工作比特只有 4 个。那么输入的量子态则为

$$|\text{Input}\rangle = |0\rangle + |4\rangle + |8\rangle + |12\rangle$$

这样就代表  $x_0 = x_4 = x_8 = x_{12} = 1$ ，并且  $\omega = e^{2\pi i/16}$ ，其它分量上都为 0。根据傅里叶变换的公式我们可以计算出

$$\begin{aligned} y_k &= \frac{1}{\sqrt{4}}(\omega^{0k} + \omega^{4k} + \omega^{8k} + \omega^{12k}) \\ &= \frac{1}{2}(1 + i^k + (-1)^k + (-i)^k) \end{aligned}$$

这里就是工作比特执行完量子傅里叶变换之后的输出态上的每个分量（第  $k$  个分量）的值。从而我们可以得到  $y_0 = y_4 = y_8 = y_{12} = \frac{1}{2}$ ，其它情况下  $y_k = 0$  ( $k \neq 0, 4, 8, 12$ )，那么最后输出的量子态则为

$$|\text{Output}\rangle = |0\rangle + |4\rangle + |8\rangle + |12\rangle$$

利用连分数分解得到周期

在最后的测量时，我们会随机得到 0, 4, 8, 12 四个结果中的一个，但是这个结果并不是周期。但是量子傅里叶变换的结果揭示了一点：

$$\omega^{irx} = e^{2\pi irx/2^N} \sim 1$$

其中我们假设测量结果是  $x$ ，总工作比特数为  $N$ ，函数的周期为  $r$ 。那么我们有

$$\frac{x}{2^N} = \frac{c}{r}$$

其中  $c$  为一个未知的整数。所以我们可以通过这个式子近似地找出函数周期。例如  $x = 4$ ， $N = 4$ ，我们有

$$\frac{c}{r} = \frac{1}{4}$$

这样我们就找到了周期  $r=4$ 。Shor 算法的量子计算机部分至此解出。你可以检验一下  $f(x) = 7^x \pmod{15}$  这个函数的周期是否确实为 4。你也可以检验一下  $f(r/2) + 1$  和  $f(r/2) - 1$  和 15 的最大公因数是否就是 15 的质因数。

有时候  $x/2^N$  并不一定能顺利约出合理的  $r$ ，这样我们就可以通过连分数分解法，得到一个逼近的分数，从而获得  $r$ 。这里就不再细讲了。

### 7.10.2 6.10.2 Shor 算法的实现

下面给出 QRunes 实现 Shor 算法的代码示例：

Python



```

@settings:
    language = Python;
    autoimport = True;
    compile_only = False;

@qcodes:
//Quantum adder MAJ module
circuit MAJ(qubit a, qubit b, qubit c) {
    CNOT(c, b);
    CNOT(c, a);
    Toffoli(a, b, c);
}

//Quantum adder UMA module
circuit UMA(qubit a, qubit b, qubit c) {
    Toffoli(a, b, c);
    CNOT(c, a);
    CNOT(a, b);
}

//Quantum adder MAJ2 module
circuit MAJ2(vector<qubit> a, vector<qubit> b, qubit c) {
    let nbit = a.length();
    MAJ(c, a[0], b[0]);
    for(let i=1: 1: nbit) {
        MAJ(b[i-1], a[i], b[i]);
    }
}

//Quantum adder, consists of MAJ and UMA modules, regardless of the carry term
circuit Adder(vector<qubit> a, vector<qubit> b, qubit c) {
    let nbit = a.length();
    MAJ(c, a[0], b[0]);
    for(let i=1: 1: nbit) {
        MAJ(b[i-1], a[i], b[i]);
    }
    for(let i=nbit-1: -1: 0) {
        MAJ(b[i-1], a[i], b[i]);
    }
    UMA(c, a[0], b[0]);
}

```

(continues on next page)

(continued from previous page)

```

}

//Determine if there is a carry
circuit isCarry(vector<qubit> a, vector<qubit> b, qubit c, qubit carry) {
    MAJ2(a, b, c);
    CNOT(b[-1], carry);
    MAJ2(a, b, c).dagger();
}

//Binding classic data with qubits
circuit bindData(vector<qubit> qlist, int data) {
    let i = 0;
    while(data >= 1){
        if(data % 2 == 1){
            X(qlist[i]);
        }
        data = data >> 1;
        i = i + 1;
    };
}

//Constant modular addition
circuit constModAdd(vector<qubit> qa, int C, int M, vector<qubit> qb, vector<qubit> qs1)
↪{
    let qNum = qa.length();
    let tmpValue = (1 << qNum) - M + C;

    bindData(qb, tmpValue);
    isCarry(qa, qb, qs1[1], qs1[0]);
    bindData(qb, tmpValue);

    circuit qCircuitTmp1;
    qCircuitTmp1.insert(bindData(qb, tmpValue));
    qCircuitTmp1.insert(Adder(qa, qb, qs1[1]));
    qCircuitTmp1.insert(bindData(qb, tmpValue));
    qCircuitTmp1.control([qs1[0]]);

    X(qs1[0]);
}

```

(continues on next page)

(continued from previous page)

```

circuit qCircuitTmp2;
qCircuitTmp2.insert(bindData(qb, C));
qCircuitTmp2.insert(Adder(qa, qb, qs1[1]));
qCircuitTmp2.insert(bindData(qb, C));
qCircuitTmp2.control([qs1[0]]);

X(qs1[0]);

tmpValue = (1 << qNum) - C;
bindData(qb, tmpValue);
isCarry(qa, qb, qs1[1], qs1[0]);
bindData(qb, tmpValue);
X(qs1[0]);
}

//Constant modular multiple
circuit constModMul(vector<qubit> qa, int constNum, int M, vector<qubit> qs1, vector
↪<qubit> qs2, vector<qubit> qs3) {
    let qNum = qa.length();

    for(let i=0: 1: qNum) {
        let tmp = constNum * pow(2, i) % M;
        circuit qCircuitTmp;
        qCircuitTmp.insert(constModAdd(qs1, tmp, M, qs2, qs3));
        qCircuitTmp.control([qa[i]]);
    }

    for(let i=0: 1: qNum) {
        CNOT(qa[i], qs1[i]);
        CNOT(qs1[i], qa[i]);
        CNOT(qa[i], qs1[i]);
    }

    let crev = modReverse(constNum, M);
    circuit qCircuitTmp1;
    for(let i=0: 1: qNum) {
        let tmp = crev * pow(2, i);
        tmp = tmp % M;
        circuit qCircuitTmp2;
        qCircuitTmp2.insert(constModAdd(qs1, tmp, M, qs2, qs3));

```

(continues on next page)

(continued from previous page)

```

        qCircuitTmp2.control([qa[i]]);
        qCircuitTmp1.insert(qCircuitTmp2);
        qCircuitTmp1.dagger();
    }
}

//Constant modular power operation
circuit constModExp(vector<qubit> qa, vector<qubit> qb, int base, int M, vector<qubit>
↪qs1, vector<qubit> qs2, vector<qubit> qs3) {
    let cqNum = qa.length();
    let temp = base;

    for(let i=0: 1: cqNum) {
        constModMul(qb, temp, M, qs1, qs2, qs3).control([qa[i]]);
        temp = temp * temp;
        temp = temp % M;
    }
}

//Quantum Fourier transform
circuit qft(vector<qubit> qlist) {
    let qNum = qlist.length();
    for (let i=0: 1: qNum) {
        H(qlist[qNum-1-i]);
        for (let j=i+1: 1: qNum) {
            CR(qlist[qNum-1-j], qlist[qNum-1-i], Pi/(1 << (j-i)));
        }
    }

    for(let i=0: 1: qNum) {
        CNOT(qlist[i], qlist[qNum - 1 - i]);
        CNOT(qlist[qNum - 1 - i], qlist[i]);
        CNOT(qlist[i], qlist[qNum - 1 - i]);
    }
}

@script:
def gcd(m,n):
    if not n:
        return m

```

(continues on next page)

(continued from previous page)

```
    else:
        return gcd(n, m%n)

def modReverse(c, m):
    if (c == 0):
        raise RecursionError('c is zero!')

    if (c == 1):
        return 1

    m1 = m
    quotient = []
    quo = m // c
    remainder = m % c

    quotient.append(quo)

    while (remainder != 1):
        m = c
        c = remainder
        quo = m // c
        remainder = m % c
        quotient.append(quo)

    if (len(quotient) == 1):
        return m - quo

    if (len(quotient) == 2):
        return 1 + quotient[0] * quotient[1]

    rev1 = 1
    rev2 = quotient[-1]
    reverse_list = quotient[0:-1]
    reverse_list.reverse()
    for i in reverse_list:
        rev1 = rev1 + rev2 * i
        temp = rev1
        rev1 = rev2
        rev2 = temp
```

(continues on next page)

(continued from previous page)

```

    if ((len(quotient) % 2) == 0):
        return rev2

    return m1 - rev2

def plotBar(xdata, ydata):
    fig, ax = plt.subplots()
    fig.set_size_inches(6,6)
    fig.set_dpi(100)

    rects = ax.bar(xdata, ydata, color='b')

    for rect in rects:
        height = rect.get_height()
        plt.text(rect.get_x() + rect.get_width() / 2, height, str(height), ha="center",
↪va="bottom")

    plt.rcParams['font.sans-serif']=['Arial']
    plt.title("Origin Q", loc='right', alpha = 0.5)
    plt.ylabel('Times')
    plt.xlabel('States')

    plt.show()

def reorganizeData(measure_qubits, quick_meausre_result):
    xdata = []
    ydata = []

    for i in quick_meausre_result:
        xdata.append(i)
        ydata.append(quick_meausre_result[i])

    return xdata, ydata

def shorAlg(base, M):
    if ((base < 2) or (base > M - 1)):
        raise('Invalid base!')

```

(continues on next page)

(continued from previous page)

```

if (gcd(base, M) != 1):
    raise('Invalid base! base and M must be mutually prime')

binary_len = 0
while M >> binary_len != 0 :
    binary_len = binary_len + 1

machine = init_quantum_machine(QMachineType.CPU_SINGLE_THREAD)

qa = machine.qAlloc_many(binary_len*2)
qb = machine.qAlloc_many(binary_len)

qs1 = machine.qAlloc_many(binary_len)
qs2 = machine.qAlloc_many(binary_len)
qs3 = machine.qAlloc_many(2)

prog = QProg()

prog.insert(X(qb[0]))
prog.insert(single_gate_apply_to_all(H, qa))
prog.insert(constModExp(qa, qb, base, M, qs1, qs2, qs3))
prog.insert(qft(qa).dagger())

directly_run(prog)
result = quick_measure(qa, 100)

print(result)

xdata, ydata = reorganizeData(qa, result)
plotBar(xdata, ydata)

return result

if __name__=="__main__":
    base = 7
    N = 15
    shorAlg(base, N)

```

C++

To Be Continue!

### 7.10.3 6.10.3 Shor 算法小结

Shor 算法首先把问题分解为了“经典计算机部分”和“量子计算机部分”。然后利用了量子态的叠加原理，快速取得了函数在一个很大范围内的取值（对于  $n$  个工作比特而言，取值范围为  $0 \sim 2^n - 1$ ）。由于函数本身是周期的，所以自变量和函数值自动地纠缠了起来，从而对于某一个函数值来说，工作比特上的态就是一组周期数态的叠加态。在取得“周期数叠加态”之后，我们自然可以通过傅里叶变换得到这组周期数的周期，从而快速解决了这个问题。

反过来看，之所以找函数周期问题能被量子计算机快速解决，是因为在工作比特上执行了一组 Hadamard 变换。它在“量子函数”的作用下，相当于同时对指数级别的自变量上求出了函数值。在数据量足够大，周期足够长的情况下，这样执行的操作总量一定会小于逐个取值寻找这个函数值在之前是否出现过——这样的经典计算机“暴力破解”法要快得多。

Shor 算法的难点在于如何通过给出的  $a$ ， $n$  来得到对应的“量子函数”形式。进一步地讲，是否存在某种方法（准确地说是具有合理时间复杂度的方法）得到任意函数的“量子计算机版本”？限于笔者知识水平不足，我只能给出目前大概的研究结论是存在某些无法表示为量子计算机版本的函数，但是幸运地是 Shor 算法属于可以表示的那一类里面。

最后，我们可以发现，量子计算机之所以快，和量子计算机本身的叠加特性有关，它使得在处理特定问题时，比如数据库搜索，比如函数求周期……有着比经典计算机快得多的方法。但是如果经典计算机在解决某个问题时已经足够快了，我们就不需要用量子计算机来解决了。

就像 Shor 算法里面所描述的那样——我们将问题分解为了量子计算机去处理的“困难问题”和经典计算机去处理的“简单问题”两个部分一样。所以，量子计算机的出现，不代表经典计算机将会退出历史舞台，而是代表着人类将要向经典计算机力所不及的地方伸出探索之手。靠着量子计算机，或许我们能提出新的算法解决化学问题，从而研制出新型药物；或许我们可以建立包含所有信息的数据库，每次只需要一瞬间就能搜索到任何问题……量子云平台是我们帮助量子计算机走出的第一步，但接下来的路怎么走，我们就要和你一同见证了。



## Chapter 8

# 第 7 章 QRunes 类型系统

至此，相信你已经对 QRunes 已经有了相对足够的认识了。对于 QRunes 更深层次的学习来说，如何理解 QRunes 的类型系统对于利用 QRunes 进行编程开发和对量子-经典混合编程的掌握是至关重要的。

QRunes 中的每一个变量，函数，函数参数，表达式都有自己的类型，类型可以确保程序编译的正常进行。

QRunes 的类型属于强类型，也是一种静态类型，即每一个对象都有自己的类型，且该类型在整个编码过程和编译运行时不会发生更改。说明：

1. 在定义变量时，量子类型和经典类型需指定具体的类型，辅助类型必须使用 `let` 关键字来指示编译器通过初始值来判定具体的辅助类型。
2. 当在定义形参变量时，所有变量必须指定其实际的类型。
3. 在声明函数时，必须指定函数的返回具体类型，如果不作指定，编译器会将该函数默认为返回值为 `qprog` 类型的量子类型函数；如果函数无返回值，则需指定函数的类型为 `void`。

在介绍 QRunes 的类型系统时，这里介绍的内容包括以下几种类型：

1. Quantum Type。类型包括：qubit
2. Auxiliary Type。类型包括：int, double 这些常规的，在量子计算框架中 Host Computer 支持的计算类型。
3. Classical Type。
4. Vector Type。
5. String Type。

## 8.1 7.1 量子类型 Quantum Type

量子类型的对象描述的是量子芯片上的量子比特，故量子类型是描述量子比特或者一组量子比特的方法。在 QRunes 中现支持 qubit 和通过 vector 构造的 `vector<qubit>`，分别表示一个量子位和一个量子位矢量。在量子器件中，量子类型的变量是到量子比特的映射，并且它可以直接用于量子门操作。如：

```
H(q); // q is a qubit.
H(qs[0]); // qs is a qvec.
```

注：H 为量子门，在 Qpanda 中有详细介绍。

量子类型另外一个常见的用法是作为量子函数中的传递参数。如：

```
fun(qubit q, vector<qubit> qs)
{
    /* use q and qs in quantum gates
    or function calls. */
}
```

在 QRunes 中，宿主语言创建了第一个量子位，底层上所有的量子码只处理量子操作。QRunes 设计者如此设计的初衷是考虑到在近期量子比特数量有限，所以考虑量子比特的动态分配和释放没有什么意义。

我们在使用这个类型时有一个限制就是在程序执行的任何时候量子比特的值都无法被改变。因此，我们只把它看作一种具有不可见状态的量子比特，只能由量子门控制，而不是一种存储量子比特数据的存储器，如振幅、相位或二进制值。量子类型的副本是引用副本，而不是值状态。赋值是量子类型出现在表达式中的唯一一种情况，因此正确的赋值表达式的语义要求我们使用正确的值创建引用或别名，如：

```
qubit alias = q; // create an alias.
H(alias); // same as H(q).
qubit q = qs[0];
qvec qs1 = qs[0:3];
```

在 QRunes，表达式 `length (qvec)` 返回辅助经典类型从而表示向量大小，因为所有 Array 类型的长度都在量子程序的编译中确定，这为上述返回值的设计提供了理论基础。

## 8.2 7.2 辅助类型 Auxiliary Type

辅助经典类型描述经典变量，这有助于我们构建量子程序。这些变量是在量子程序传输到量子设备之前确定的。在 QBuilder 系统中，未正式考虑辅助类型，相反，这些变量的处理由宿主语言或编程接口维护。这里我们只处理普通的数值类型和数组类型。如：

```
let i = 0;
H(q[i]);
```

注：变量 i 的赋值由宿主语言处理并且它不传输到量子计算机。

另一个例子是对所有量子比特进行迭代并执行 hadamard 门。如下：

```
for (i = 0 : len(q))
    H(q[i]);
```

如果宿主语言支持动态类型系统（例如 JavaScript），或支持类型或隐式类型（如 C++ 和 Python），那么我们只需要使用 let 关键字来定义辅助类型。例子如下：

```
let i = 0;
let b = 0.1;
RX(q[i],b);
```

### 8.3 7.3 经典类型 Classical Type

经典类型处理量子计算机中的经典位或变量。在物理底层实现中，量子芯片并不单独工作，这是因为控制和测量信号由一组嵌入式设备处理，测量结果暂时保存在这些设备的内存中。本设计旨在实现量子程序的反馈控制，如 qif 和 qwhile。量子比特是脆弱的，在纳秒到微秒的时间内会崩溃，这比量子比特系统执行反馈所需的时间要短得多。这要求我们使用嵌入式器件来实现量子芯片控制从而满足系统的反馈要求。

经典类型是一种在嵌入式系统中表示数据的专用方法，cbit 是经典类型一个例子，它用于保存量子位的测量结果。cvec 是一个 cbit 数组，例子如下：

```
measure(q,c);    // perform a measurement.
for (i = 0:1: q.length())    // measuring a qvec to a cvec.
    measure(q[i],c[i])
```

提取经典类型通常涉及与将数据从嵌入式系统传输到主机的嵌入式系统的驱动接口，这点由宿主语言就可以直接处理，与量子类型类似，经典类型被认为是对嵌入式系统内存的一种映射，它不能通过赋值直接修改值。此外，分配和释放不被视为 QBuilder 系统的一部分。经典类型可以出现在量子程序的反馈控制序列中，例如量子选择和迭代。这一部分将在量子经典反馈部分介绍（目前没有）。经典类型可以出现在算术表达式中，与经典类型的变量相关的表达式不允许在量子计算机中进行计算，只在上述嵌入式系统中进行处理。这种类型的表达式（我们称之为经典表达式）具有量子门的等效位置。处理具有经典类型的表达式与辅助表达式的处理方式不同，这一部分将在表达式章节介绍。经典表达式的返回值是经典类型变量，对经典类型和辅助类型的二元操作符操作是有效的，返回值是经典类型。下面是一个例子：

```
C1 = C2;
C2 = !C2;
qif (C2) { // do something }
qif (C1) { // C1 is the negative of C2 }
```

将常量赋给经典类型也是有效的，但我们必须小心，因为程序在传输到量子计算机之前，实际上并没有执行分配操作。反过来，辅助类型不能由经典类型指定，下面是一个例子：

```
C1 = True
qif (C1) { // do something }
let a = C1; // Bad assignment
```

### 7.4 向量类型 Vector Type

向量类型是一种数据结构，表示可以代表（存储）多个同种类型的对象的集合，并可以根据索引来进行检索这些对象。向量类型的对象类型为其组成中的对象的基本类型。向量类型的在构造时不需要指定其长度。

向量类型的构造方式：

1. `vector <T> ivec;` // T 可以是 AQC 系统中的任何类型。
2. `let ivec = [1,2,3];` // 该方式仅可以支持辅助类型（Auxiliary Type）。

Vector 类型的用法及函数调用：

1. 通过下标获取向量中的元素：`Iveco[i]`。
2. `length()`：返回向量的长度。
3. `append (t)`：在向量的末尾增加一个元素。
3. `pop()`：删除向量中的最后一个元素。

### 7.5 字符串类型 String Type