量易伏用户手册

百度量子计算研究所

2022 年 7 月 10 日

目录

前言

阅读这本手册前，请先浏览下面的章节结构快速总结。

- 第一章是量易伏简介，帮助用户对量易伏产生全局的理解。用户在阅读这一部分时感到困惑的地方可以先略过，因为量易伏是一个量子计算编程平台，只有实践才能使用户真正了解量易伏并让量易伏成为自己强大又便利的工具。
- 第二章和第三章是使用量易伏的入门教程，适用于需要快速上手使用量易伏的用户。看完这一部分用户将对量易伏的四大前端和多个后端的使用方式产生基本了解，并且能够使用量易伏的在线方式和本地方式进行实验。
- 第四章帮助用户更深层次地使用量易伏，本章所提供的量子计算示例均已加入 QCompute / PyOnline / YunIDE 的电路模板，用户可以在这些前端平台直接使用。
- 第五章为 API 相关文档。
- 附常见问题和参考文献。

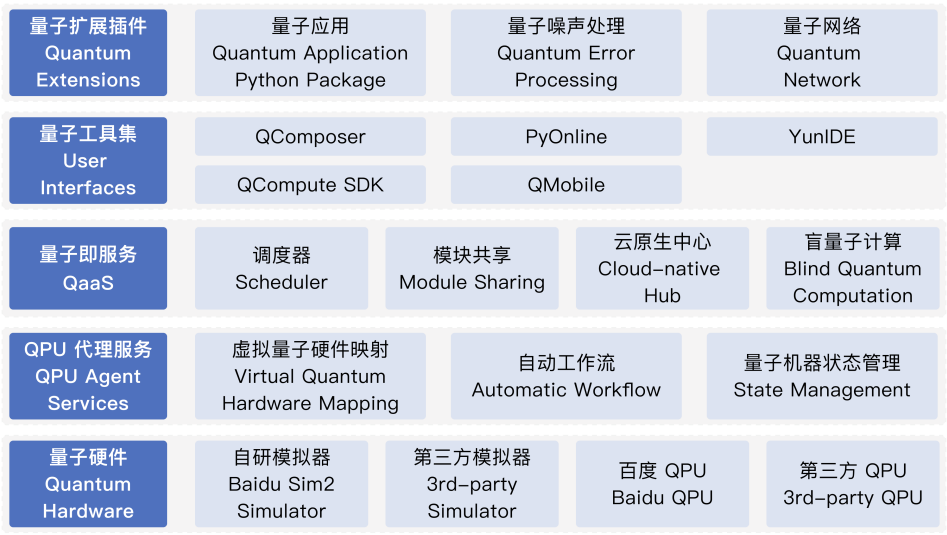
欢迎用户通过 Quantum-hub 的用户反馈或者百度量易伏用户群进行提问交流。

群名称:百度量易伏用户群
群　号:1147781135

第一章　量易伏简介

量易伏 (英文名: Quantum Leaf) 是百度研究院旗下量子计算研究所开发的云原生 (Cloud Native) 量子计算平台，提供以量子设施即服务 (Quantum infrastructure as a Service 简写为 QaaS) 模式的量子计算环境。量易伏为用户提供了四种方式进行量子计算实验，其中 QCompute 支持本地运行和连接在线资源运行，PyOnline、QComposer 和 YunIDE 则需要登录 Quantum-hub 在线运行。

1.1　用户端-量子端架构



量易伏架构
Quantum Leaf Architecture

1. 用户端可以选用四类不同组件实现编程

   - **直观的 QComposer**: QComposer 是可视化的量子编程工具。通过拖动量子门图标和 QASM 语言编程相结合的方式，用户可以在 QComposer 上完成量子电路的搭建和运行。
   - **简单的 PyOnline**: PyOnline 是部署在云端的轻量级量子编程开发环境。用户可以在 PyOnline 上使用 Python 和 QASM 语言混合编程。PyOnline 含有丰富的量子电路模板，即开即用。
   - **强大的 YunIDE**: YunIDE 是基于 code-server 开发的，部署在云端的全量级量子编程开发环境。通过设置 cpu 参数和内存参数并选择镜像环境来使用的云端集成开发环境，默认携带 Jupyter Notebook 文档 (.ipynb) 的渲染支持。
   - **灵活的 QCompute**: QCompute 是基于 Python 的开源 SDK，支持 Python 和 QASM 混合语言编程。用户需下载安装 QCompute 并进行相应环境配置才能使用。由于携带本地模拟器，QCompute 支持量子电路的本地运行。
   - **便捷的 QMobile**: 一个可用于学习量子计算并进行量子计算实验的移动端应用。

2. 量子端有本地模拟器、在线模拟器和 QPU 供选用

   - 本地模拟器使用本地计算资源，用户可以在无网络的情况下使用。
   - 在线模拟器使用百度云计算资源，用户需传入自己的 Token 才能使用，计算电路需消耗 Credit。
   - QPU 通过代理（QAgent）提供服务，**VIP 用户有优先享有 QPU 测试使用的权利**。

1.2 用户端

用户端包含四大套件，解决不同用户的需求。选用得心应手的用户端，可以让用户的工作事半功倍。作为云原生的应用，这些用户端的使用和研发，都离不开量子中控平台 Quantum-hub 的搭配使用。

1.2.1 量子中控平台: Quantum-hub

Quantum-hub 是量子计算的核心平台，支持起了 QaaS (Quantum infrastructure as a Service) 完整体验。从 Quantum-hub 可以到达各个用户端，QComposer / PyOnline / QCompute SDK / YunIDE 的结果都可以从 Quantum-hub 查看或下载。用户还可以从 Quantum-hub 获取使用 QCompute 所需的 Token，改变任务提交的状态以及查看提交的电路等。

1.2.2 量子作曲家: QComposer

QComposer 是一个可视化的在线量子电路编辑环境。用户拖动门图标即可创建电路并在左侧代码框生成相应 QASM 代码，编辑 QASM 代码同样实时自动将电路绘制在右侧。用户无需进行复杂的配置，登陆 Quantum-hub 即可使用。

1.2.3 量子在线平台: PyOnline

PyOnline 是支持 Python 语言的 web 编辑器,同时也是一个完备的 Python 代码托管环境。用户可以在创建项目时选择环境，但是无法修改环境预定义的设置。用户可以通过文件树的上传下载按钮来导入数据及获取结果,还可以通过内置选项卡设置所依赖的 pip 包列表,PyOnline 将自动生成相应 requirements.txt 文件。

1.2.4 量子工作站: YunIDE

YunIDE 是一个定制化的云端集成开发环境，基于 code-server 开发的全功能 IDE。YunIDE 基础环境默认携带 Jupyter Notebook 文档 (.ipynb) 的渲染支持，用户通过选择基础环境和参数构建自己的代码空间后即可使用。区别于 QCompute、PyOnline 和 QComposer 的是: YunIDE 需要创建并启动环境才能使用，开启 YunIDE 环境每隔 1 小时消耗一定点数（Credit, 参考2.1.3,5），消耗速率与参数设置（CPU 和内存）直接相关，扣点数标准如下（单位: 点/h）:

费率 = CPU 核心数 * (内存 / 512M)

1.2.5 量子开发套件: QCompute SDK

QCompute SDK 编程框架使用 Python 3 开发，并封装为二次开发包形式发布。用户可编写 Python 3 程序，调用本开发包，设计量子计算应用程序。QCompute 含有丰富的量子电路模板和教程，用户可以在 GitHub 上下载安装 QCompute，也可以使用语句 `pip install qcompute` 安装 QCompute。

1.2.6 量子移动端: QMobile

QMobile 是一个可用于学习量子计算并进行量子计算实验的移动端应用。QMobile 无需任何编程经验，即开即用，方便初学者体验和学习量子计算。QMobile 为开发者提供便捷的量子计算任务和量易伏账号管理功能，并用户指南和图文反馈渠道加强对用户的技术支持。

1.3 量子端

- 本地模拟器: 量易伏提供 1 个本地模拟器，设备标识符为 LocalBaiduSim2。本地模拟器无需将量子任务提交到云端，使用本地设备资源即可完成计算，支持用户在无网络的情况下使用。
- 云端模拟器: 量易伏提供 7 个云端模拟器，其中水和气支持多任务并行计算；土、雷，天和湖算力强劲；风支持稀疏模式。云端模拟器使用百度云计算资源，用户需传入自己的 Token 才能使用，计算电路需消耗 Credits。
- 量子计算机: 量易伏提供 3 个量子计算机，其中 CloudIoPCAS 和 CloudBaiduQPUQian 是分别来自中国科学院物理所和百度量子计算研究所的超导量子计算机；CloudIonAPM 是来自中科院精密测量科学与技术创新研究院的离子阱量子计算机。量子计算机通过百度云计算资源提供服务，用户需传入自己的 Token 才能使用，计算电路需消耗 Credits。VIP 用户有优先享有量子计算机测试和使用的权利。

量易伏提供的量子端如下表所示，更多量子计算机参数和状态，以及量子测量噪声处理等相关信息，请在服务状态页，点击对应卡片查看。

类型	量子端	说明
本地	LocalBaiduSim2	使用 Python 编写的 Sim2 本地版
云端	CloudBaiduSim2Water	使用 C++ 编写的多实例 Sim2 模拟器云版
	CloudBaiduSim2Earth	使用 Python 编写的单一实例高配置 Sim2 模拟器云版
	CloudBaiduSim2Thunder	使用 C++ 编写的单一实例高配置 Sim2 模拟器云版
	CloudBaiduSim2Wind	使用 C++ 编写的单一实例 Sim2 模拟器云版（支持稀疏模式）
	CloudBaiduSim2Heaven	使用 C++ 编写的单一实例集群 Sim2 模拟器云版
	CloudBaiduSim2Lake	使用 C++ 编写的单一实例 Sim2 模拟器云版 (支持 GPU 运算)
	CloudAerAtBD	开源 Aer(C++ 版) 模拟器云版
	CloudIoPCAS	来自中科院物理所的 10 比特超导量子计算机 (VIP)
	CloudBaiduQPUQian	来自百度量子计算研究所的 8 比特超导量子计算机 (VIP)
	CloudIonAPM	来自中科院精密测量院的 1 比特离子阱量子计算机 (VIP)

注: 在 QCompute 使用云端模拟器和 QPU 时需传入 Token（参考3.2.2）。

1.3.1 BaiduSim2

Sim2 是百度自主研发的量子模拟器产品，使用 Tensor contraction 同构算法，具有较高性能和稳定表现。Sim2 分为本地版和云版两个版本，分别通过使用不同的后端名声明调用，例如:

- env.backend(BackendName.LocalBaiduSim2)
- env.backend(BackendName.CloudBaiduSim2Water)
- env.backend(BackendName.CloudBaiduSim2Earth)
- env.backend(BackendName.CloudBaiduSim2Thunder)
- env.backend(BackendName.CloudBaiduSim2Wind)
- env.backend(BackendName.CloudBaiduSim2Heaven)
- env.backend(BackendName.CloudBaiduSim2Lake)

注: 用户可以指定 BaiduSim2 模拟器的各项参数（参考3.2.3）。

1.3.2 Aer

Aer 是著名开源模拟器产品，有一定的用户积累，我们将其部署在百度云上供用户使用。用户可以通过声明 env.backend(BackendName.CloudAerAtBD) 调用。QCompute 使用 Aer 模拟器需在 Quantum-hub 上注册用户，并传入 Token。

1.3.3 QPU

QPU 通过代理（QAgent）提供服务。使用 QPU 需注册 Quantum-hub，获取 VIP 权限，并传入 Token。量易伏提供 3 个量子计算机，其中 CloudIoPCAS 和 CloudBaiduQPUQian 是分别来自中国科学院物理所和百度量子计算研究所的超导量子计算机；CloudIonAPM 是来自中科院精密测量科学与技术创新研究院的离子阱量子计算机。用户可以通过 Quantum-hub 的服务状态页，了解 QPU 的实时状态信息。通过如下声明使用:

- env.backend(BackendName.CloudIoPCAS)
- env.backend(BackendName.CloudBaiduQPUQian)
- env.backend(BackendName.CloudIonAPM)

Available Qubits	f10 (GHz)	Ec (GHz)	T1 (us)	T2 (us)	ReadoutFrequency (GHz)	PiLen (ns)	SingleQubitGateFid	measureFid	Available CZgate	CZFid	CZLength (ns)
q1	5.492	0.250	43.8	19.0	6.664	30.0	98.45%	[98.5%, 94.4%]	q1q2	96.80%	41.45
q2	4.497	0.207	52.9	1.6	6.646	30.0	99.68%	[97.2%, 92.0%]	q2q3	94.50%	39.12
q3	5.420	0.251	21.7	2.5	6.628	30.0	99.21%	[98.5%, 94.9%]	q3q4	92.30%	36.00
q4	4.444	0.206	39.8	1.9	6.608	30.0	99.04%	[96.5%, 90.4%]	q4q5	94.20%	35.00
q5	5.468	0.251	14.9	9.1	6.593	30.0	99.69%	[98.2%, 92.7%]	q5q6	93.70%	35.00
q6	4.393	0.203	32.0	1.2	6.571	30.0	99.14%	[97.2%, 93.3%]	q6q7	95.60%	36.35
q7	5.384	0.252	28.8	7.2	6.554	30.0	98.07%	[98.1%, 92.3%]	q7q8	94.20%	37.57
q8	4.358	0.203	26.0	1.1	6.532	30.0	99.81%	[96.9%, 91.3%]	q8q9	92.40%	31.50
q9	5.298	0.246	24.1	2.1	6.511	30.0	99.94%	[96.7%, 89.1%]	q9q10	94.30%	31.50
q10	4.402	0.208	24.8	1.0	6.490	30.0	99.90%	[96.5%, 92.5%]			

上表来自 Quantum-hub 服务状态页，刻画 CloudIoPCAS 的各项指标，各参数释义如下:

1. 单量子比特的属性:
   - AvailableQubits - 可用的单量子比特列表
   - f10 - 量子比特的频率，即基态 $|0\rangle$ 和激发态 $|1\rangle$ 之间的能量差，单位 GHz
   - ReadoutFrequency - 读取量子比特状态所使用的读取脉冲频率，单位 GHz
   - Ec - 量子比特的非谐性，单位 GHz
   - T1 - 量子比特的振幅退相干时间，单位 us
   - T2 - 量子比特的相位退相干时间，单位 us
   - SingleQubitGateFid - 单量子比特逻辑门的平均保真度
   - PiLen - 单量子比特逻辑门操控的门长，单位 ns
   - measureFids - 量子比特 $|0\rangle$ 态和 $|1\rangle$ 态的读取保真度
2. QPU 的其他信息:
   - AvailableCZgate - 可用的双量子比特 CZ 门列表
   - CZFid - 双量子比特 CZ 门的保真度

- CZLength - 双量子比特 CZ 门操控的门长，单位 ns

Available Qubits	Ion	T1 (s)	T2 (ms)	PiLens (us)	SingleQubitGateFid (%)	SPAMFid (%)
Q1	CA40	1.2	0.5	2.5	96.90%	100.00%

上表来自 Quantum-hub 服务状态页，刻画 CloudIonAPM 的各项指标，各参数释义如下:

1. 单量子比特的属性:

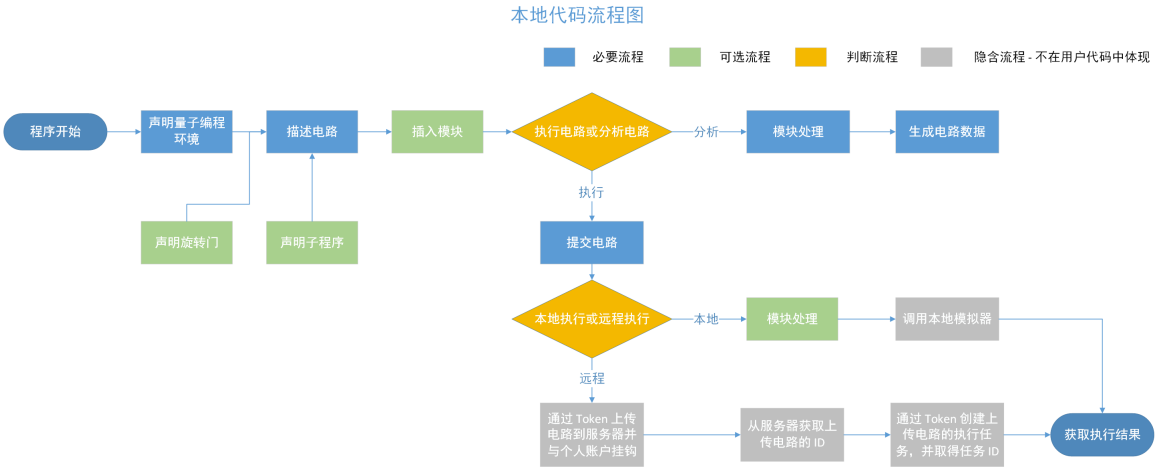
   - AvailableQubits - 可用的单量子比特列表
   - Ion - 离子种类
   - T1 - 量子比特的能级驰豫时间，单位 s
   - T2 - 量子比特的相位退相干时间，单位 ms
   - PiLen - 单比特量子逻辑门的时间长度，单位 us
   - SingleQubitGateFid - 单量子比特逻辑门的平均保真度
   - SPAMFid - 态制备和读取保真度

1.4 相关流程

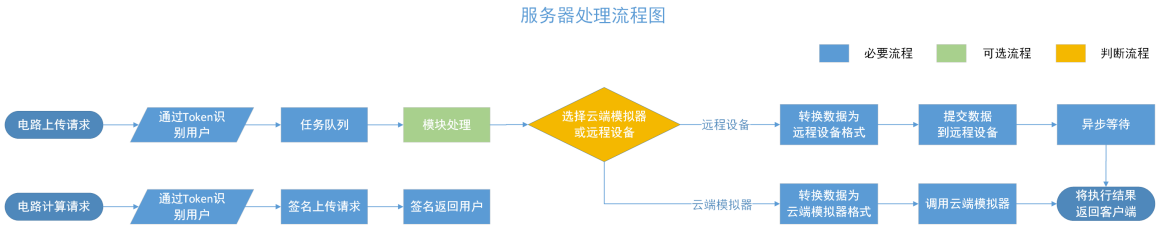
本小节提供的技术说明或许能为感兴趣的用户提供一些解答，暂时无法理解这些流程图的用户可以先忽略这一小节，待对量易伏有更多的实践经验并对其流程产生兴趣时再看不迟。

1.4.1 QCompute 相关

- QCompute 的本地代码流程

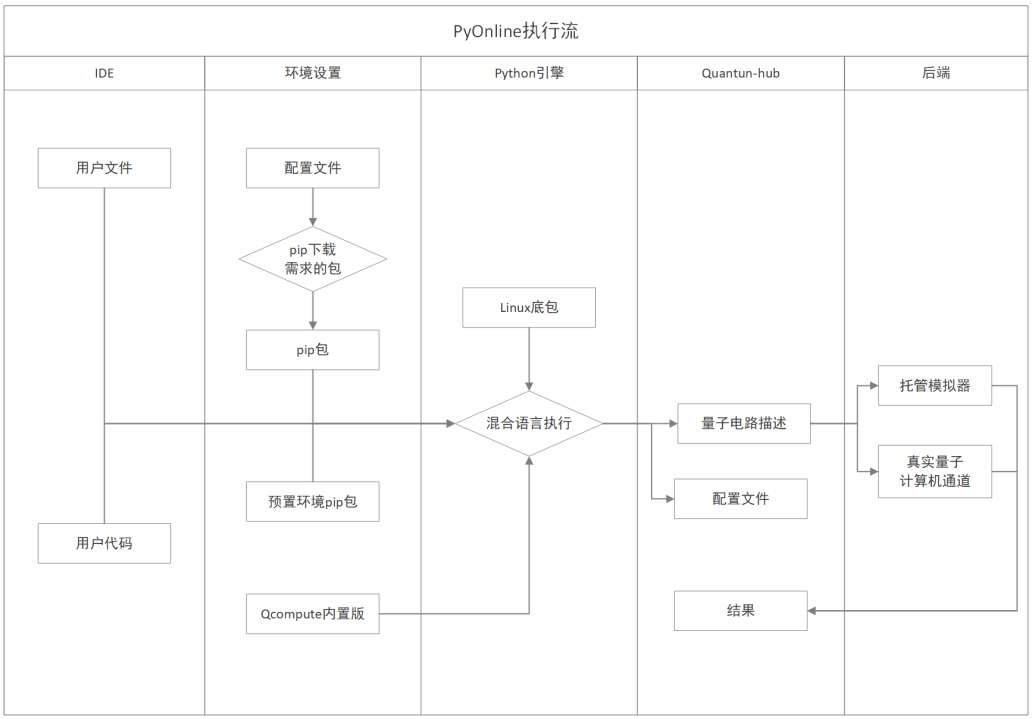


- QCompute 的服务器处理流程



1.4.2 PyOnline 相关

- PyOnline 的执行流

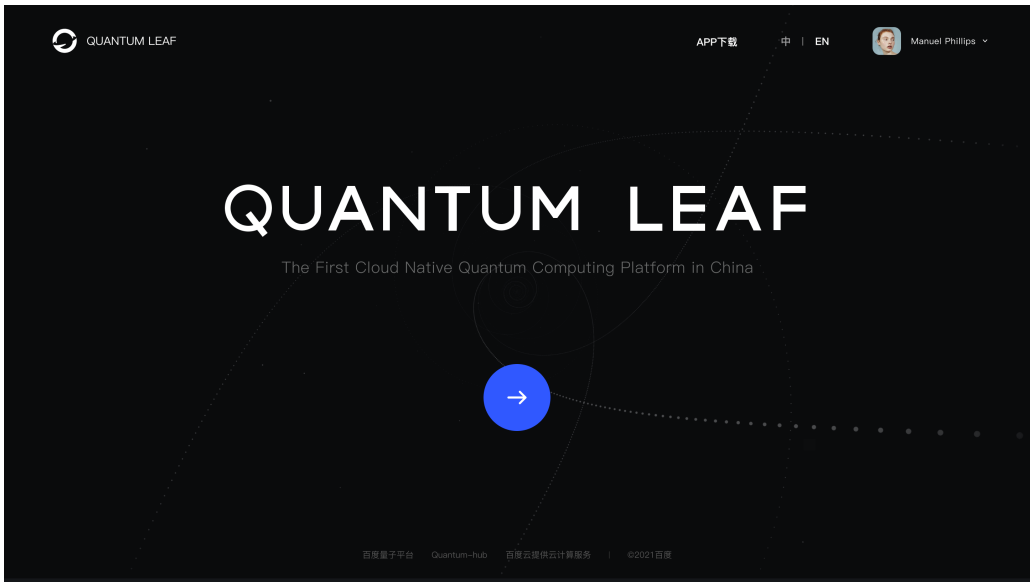


第二章　在线使用量易伏

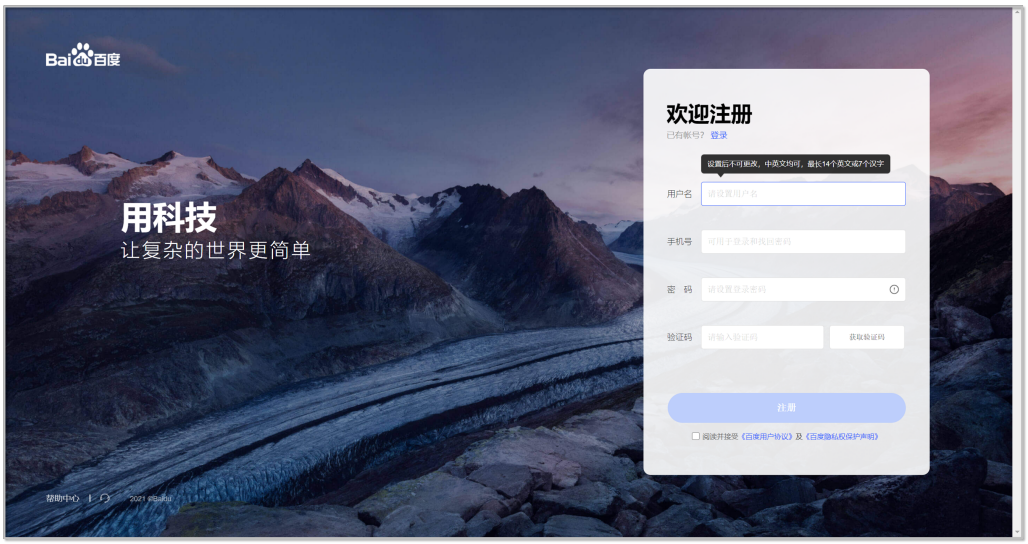
2.1　账号管理

2.1.1　注册与登录

新用户请使用浏览器访问 Quantum-hub，使用百度帐号或 Github 账号登陆量易伏。

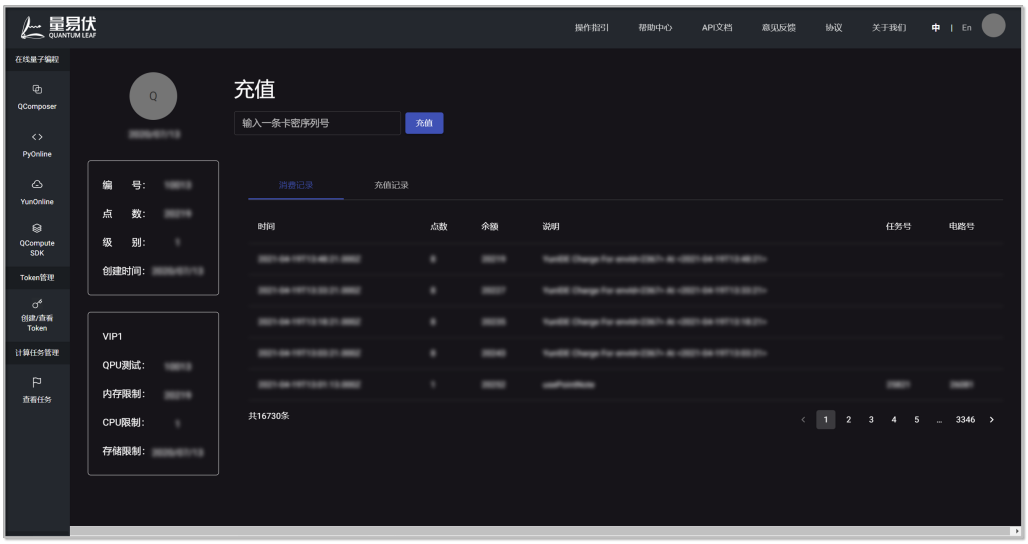


如无百度账号，点击【注册/登陆】-【立即注册】跳转到如下页面注册百度账号。



2.1.2 查看用户信息

点击头像下拉菜单【个人中心】可查看账户信息，包括编号、昵称 (可修改)、点数、创建时间等。



2.1.3 点数相关

用途

- 每次使用云端模拟器运行量子电路消费 1 点;
- 使用 YunIDE 根据资源和用时也将消费一定点数，消耗速率详见 1.2.5;
- 每次使用真机运行量子电路消费 1 点。

获得途径

- 新用户赠送 128 个点数;
- 通过 Quantum-hub 内的【意见反馈】申请免费点数。免费点数将以卡密序列号形式发送到用户邮箱。用户在【个人中心】界面的【充值】栏输入卡密序列号获得对应点数，每个卡密序列号只能使用一次;
- 使用实体充值卡，手机扫码获得点数。实体卡获得途径请关注百度量子的讲座、会议、赛事等各类活动。

其他

- 点数不足时将无法在线运行量子电路;
- 用户使用 QCompute 的 LocalBaiduSim2 模拟器运行电路时不消耗点数;
- 点数的充值记录和消费记录请在【个人中心】查看。

2.1.4 生成 Token

Token 是用户使用云端量子计算资源的授权凭据，其形式是一串 24 位字符，展示在【创建/查看 Token】页。用户注册量易伏账号时生成默认 Token，Token 也可以由用户另行创建。一个 Token 属于用户个人所有，由用户创建并管理。

2.2 PyOnline

PyOnline 是一个方便的在线量子计算开发环境，用户无需进行复杂的配置，登陆 Quantum-hub 后即可使用，达到快速使用量易伏进行实验的目的。

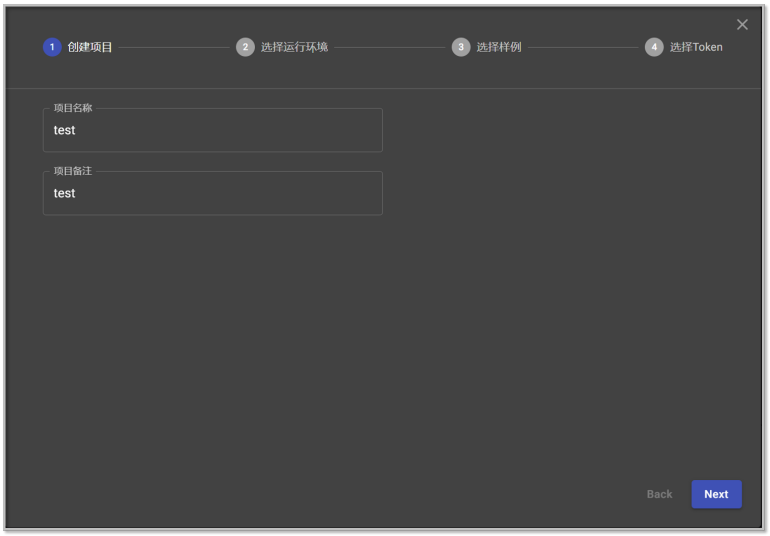
2.2.1 熟悉界面

工具栏说明如下:

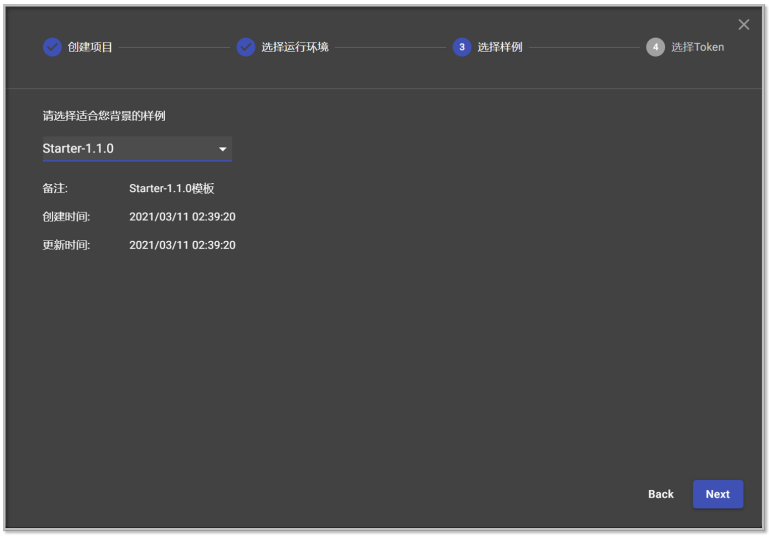
工具栏按钮	功能描述	工具栏按钮	功能描述
	新建项目		查看项目属性，可编辑项目名等
	查看本电路的运行历史		强制停止运行中的电路
	保存当前打开的文件		运行电路
	关闭当前打开的文件		

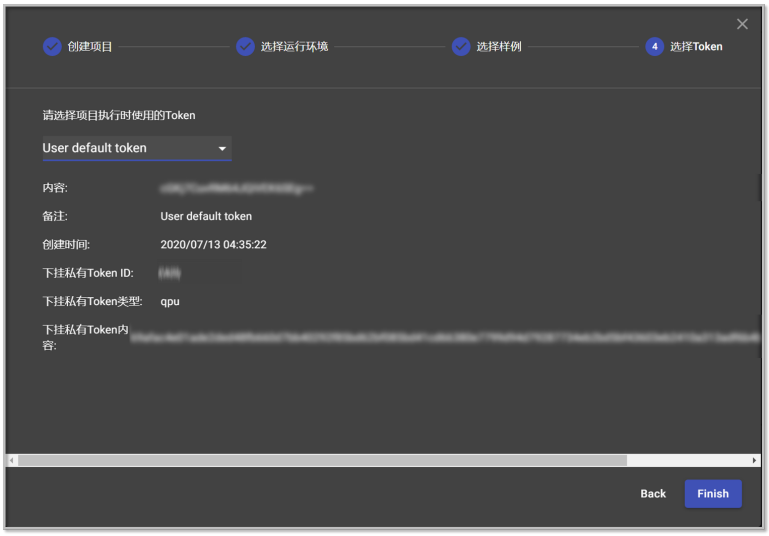
2.2.2 创建项目

点击【新建】按钮，跟随项目创建引导创建项目，该引导共分四步:



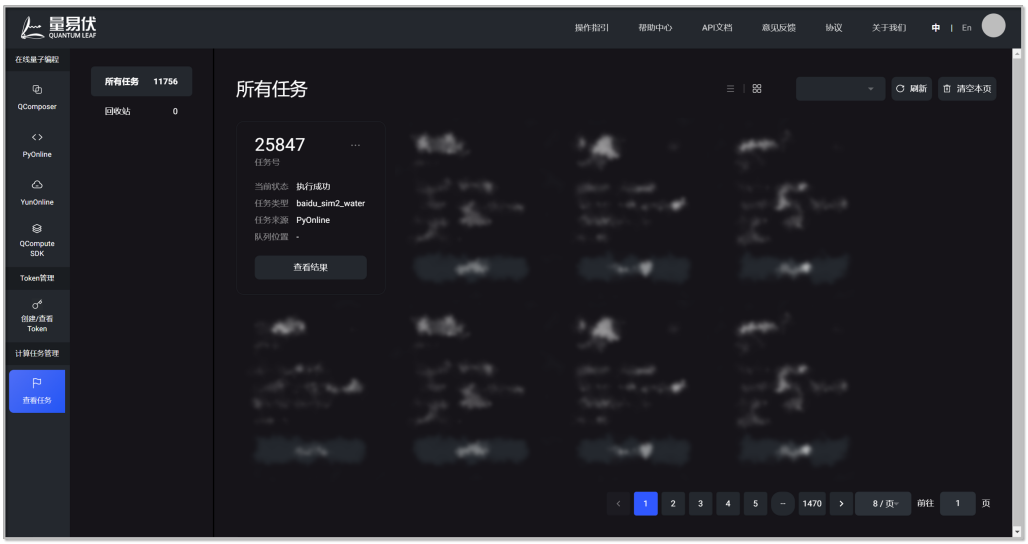

创建项目
2 选择运行环境
3 选择样例
4 选择Token
请选择所需的Python运行环境
QCompute SDK 1.1.0
备注: QCompute 1.1.0
状态: enabled
Python版本: 3.8.7
创建时间: 2021/03/11 02:08:30
更新时间: 2021/03/11 09:21:46
包 要求 版本
bce-python-sdk == 0.8.59
bidict == 0.21.2
Back
Next

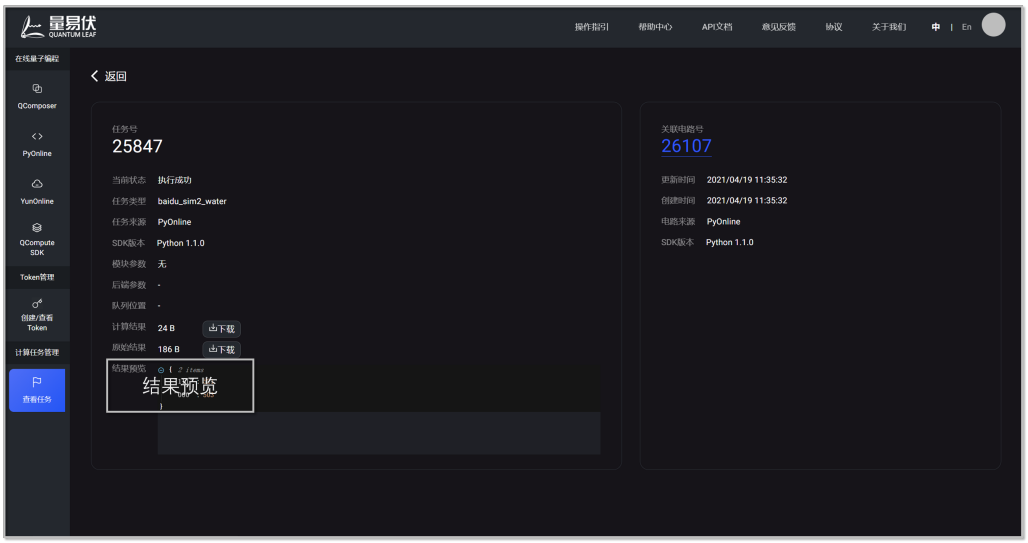

创建项目
选择运行环境
3 选择样例
4 选择Token
请选择适合您背景的样例
Starter-1.1.0
备注: Starter-1.1.0模板
创建时间: 2021/03/11 02:39:20
更新时间: 2021/03/11 02:39:20
Back
Next


创建项目
选择运行环境
选择样例
4 选择Token
请选择项目执行时使用的Token
User default token
内容:
备注: User default token
创建时间: 2020/07/13 04:35:22
下挂私有Token ID:
下挂私有Token类型: qpu
下挂私有Token内容:
Back
Finish

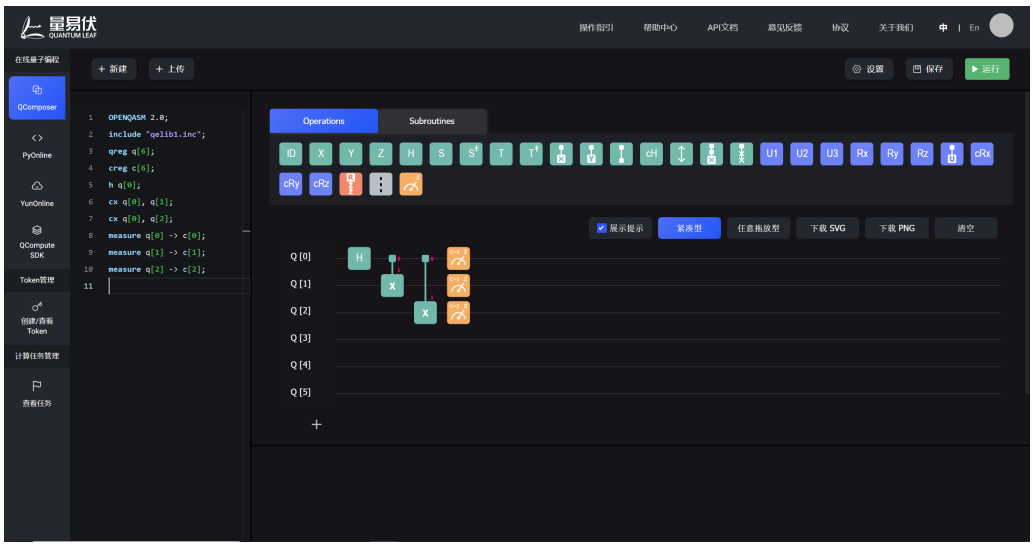
2.2.3 查看结果

用户可以在【查看任务】界面查看或下载计算结果:





还可以点击电路号跳转到 QComposer 界面查看电路图:



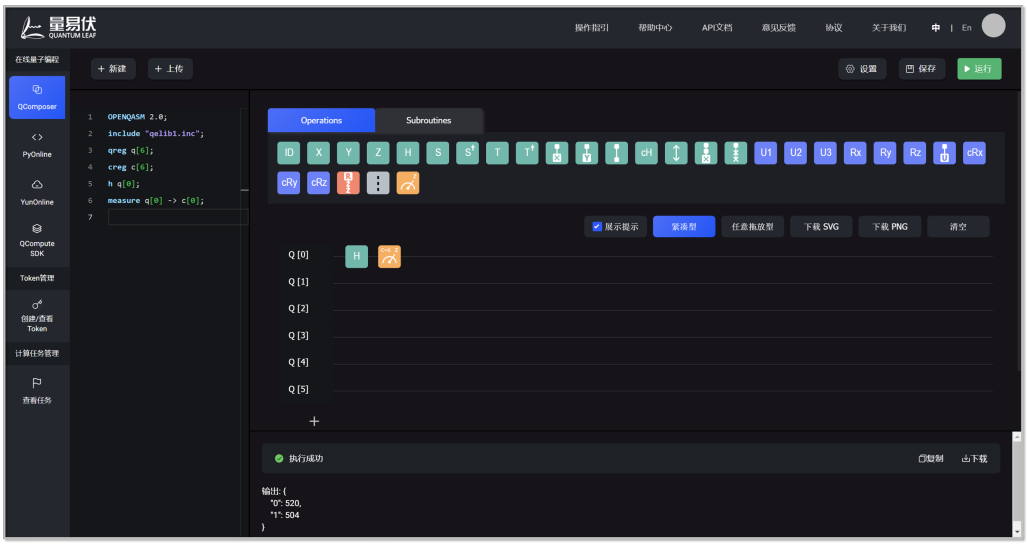
2.3 QComposer

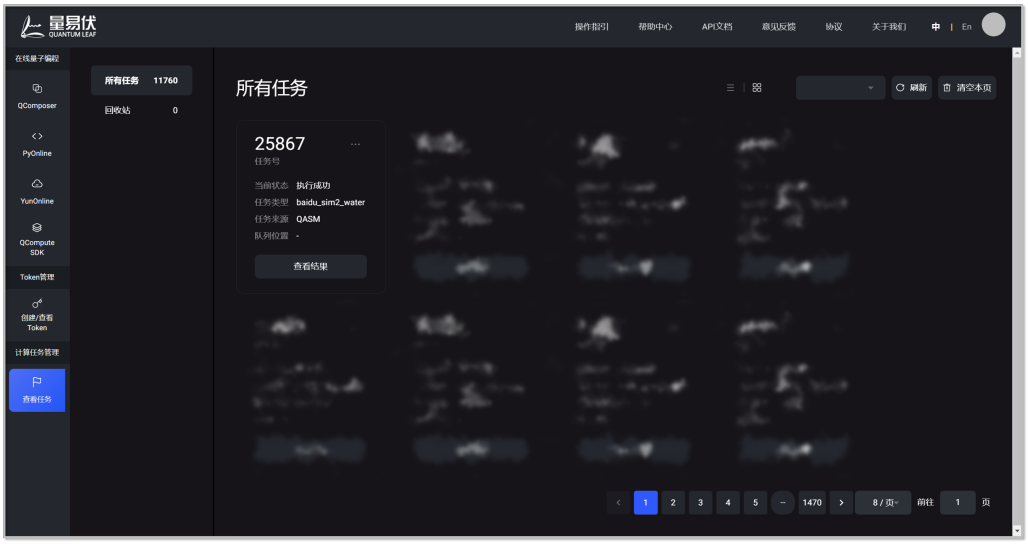
2.3.1 熟悉界面

QComposer 是一个可视化的在线量子电路编辑环境。用户拖动门图标即可创建电路并在左侧代码框生成相应代码。代码行注释符为 '//'。

2.3.2 查看结果

QComposer 运行结果可以在执行结果窗口预览，也可以在【查看任务】处查看和下载:





2.4 YunIDE

2.4.1 熟悉界面

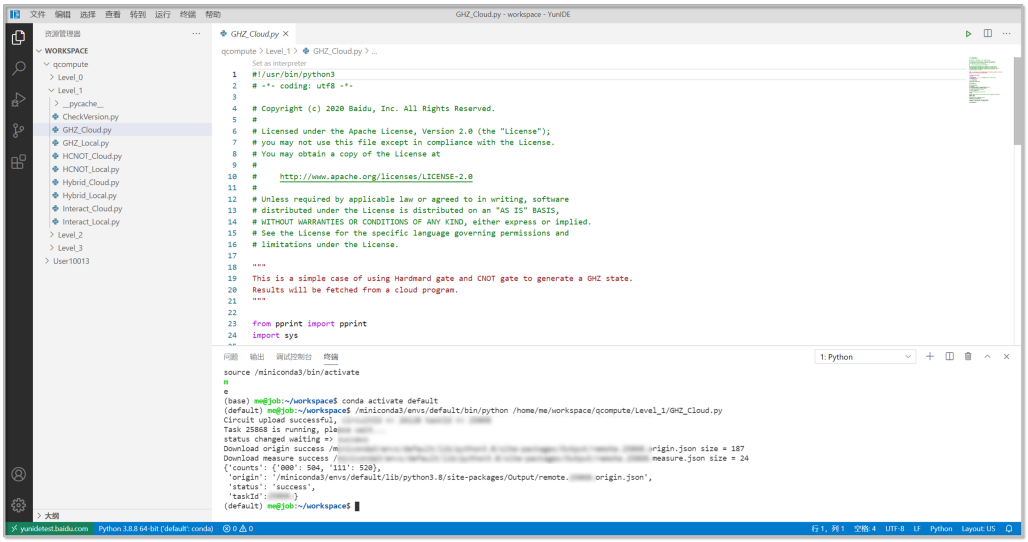
YunIDE 是一个集成开发环境，窗口主要分为资源管理器，代码编辑器和交互面板，以及各类菜单。

2.4.2 创建环境

开启 YunIDE 将消耗点数。在使用的过程中，用户可以修改虚拟环境的 CPU 和内存参数。建议用户先使用 1-2 核，1024M 配置的虚拟环境进行学习和实践，当有需要时再行扩充。

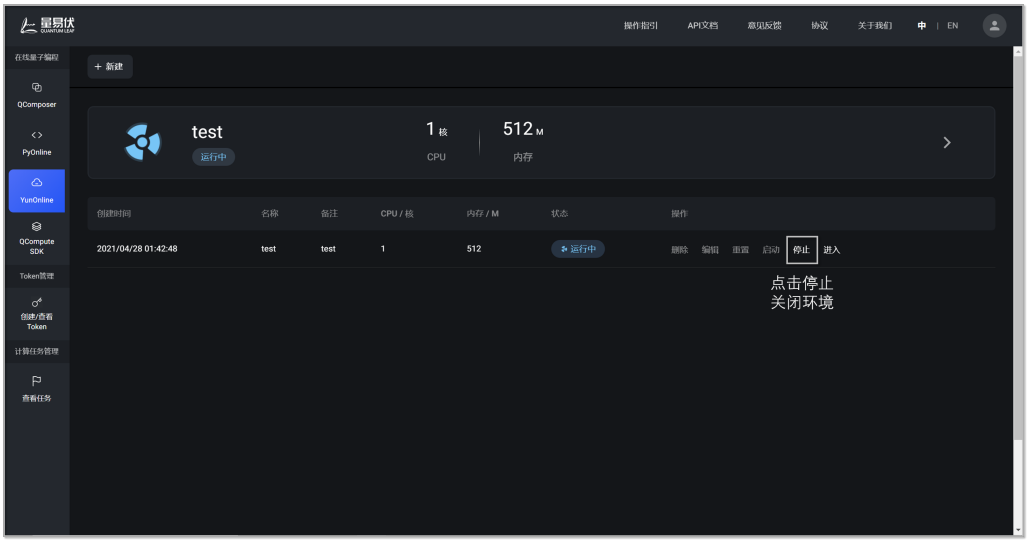
2.4.3 运行项目

Python 虚拟环境可以使用默认的，无需用户配置。如需安装依赖包，用户在终端使用 pip 即可完成。除了运行示例代码外，用户也可以自行创建项目文件，同样是可行的。YunIDE 上运行的项目结果可以在 YunIDE 直接预览和下载，也可以到【查看任务】进行查看和下载。



2.4.4 关闭环境

停止环境后，环境中数据将被清空。关闭环境应在 Quantum-hub 的 YunIDE 界面完成。用户也可以设置自动休眠时间，从关闭 IDE 界面开始倒计时，倒计时完成便关闭 IDE，并停止扣除点数。如果用户始终保持 IDE 界面开启，将因为 YunIDE 没有正常关闭而继续扣费，直到点数扣完系统才会自动关闭环境。


量易伏
操作指引
API文档
意见反馈
协议
关于我们
中
EN
在线量子编程
QComposer
PyOnline
YunOnline
QCompute SDK
Token管理
创建/查看 Token
计算任务管理
查看任务
+ 新建
test
运行中
1 核
CPU
512 M
内存
创建时间
名称
备注
CPU / 核
内存 / M
状态
操作
2021/04/28 01:42:48
test
test
1
512
运行中
删除
编辑
重置
启动
停止
进入
点击停止
关闭环境

第三章　本地使用量易伏

QCompute 是一个可以在本地独立使用的量易伏量子计算二次开发工具，定位为基于 Python 的 SDK (Python SDK)，可以通过使用远程量子端调用在线的计算资源。**请注意**，使用 QCompute 调用在线资源 **必须** 事先登录 Quantum-hub 生成并拷贝 Token，粘贴到运行代码的 `Define.hubToken = ""` 。

3.1　QCompute 安装

3.1.1　安装 Python

请使用 Python 3.7 - 3.9 版本的运行环境。

3.1.2　安装 QCompute

安装 QCompute 有以下两种方式，用户可任选其一。

方式一: 从源代码安装 QCompute

1. 从 GitHub 下载源代码包并解压缩
2. 在源码目录执行 `pip install -e .`

方式二: 从 PyPI 安装 QCompute

执行 `pip install qcompute`

3.1.3　验证安装

- 安装完成后，运行

```
python -m Test.PostInstall.PostInstall_test
```

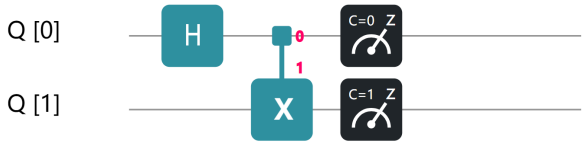
按照提示输入 Token，稍候片刻，如输出:

```
Local test succeeded.
Cloud test succeeded.
```

表示测试通过。

3.2　QCompute 示例

此节以本地计算和远程计算两类分别展示使用 H 门和 CX 门构建 Bell 态。构建 Bell 态电路是量子计算中的“Hello World”，是一个经典的量子计算入门电路。



3.2.1 本地计算

- 引入所需要的包

```
from pprint import pprint
import sys
sys.path.append('../..')
from QCompute import *
```

- 创建量子编程环境

```
env = QEnv()
```

- 选择 LocalBaiduSim2 作为量子端

```
env.backend(BackendName.LocalBaiduSim2)
```

- 编写量子电路

```
q = env.Q.createList(2)
H(q[0])
CX(q[0], q[1])
MeasureZ(*env.Q.toListPair())
```

- 提交并查看结果

```
taskResult = env.commit(1024, fetchMeasure=True)
pprint(taskResult)
```

- 任务结果

```
{'00': 513, '11': 511}
```

3.2.2 远程计算

- 引入所需要的包

```
from pprint import pprint
import sys
sys.path.append('../..')
from QCompute import *
```

- 输入 Token

```
Define.hubToken = "填入␣Token"
```

- 创建量子编程环境

```
env = QEnv()
```

- 选择 CloudBaiduSim2Water 作为量子端

```
env.backend(BackendName.CloudBaiduSim2Water)
```

- 编写量子电路

```
q = env.Q.createList(2)
H(q[0])
CX(q[0], q[1])
MeasureZ(*env.Q.toListPair())
```

- 提交并查看结果

```
taskResult = env.commit(1024, fetchMeasure=True)
pprint(taskResult)
```

- 任务结果

```
{'00': 510, '11': 514}
```

此处选择 CloudBaiduSim2Water 作为量子端进行远程计算，更多选项如下:

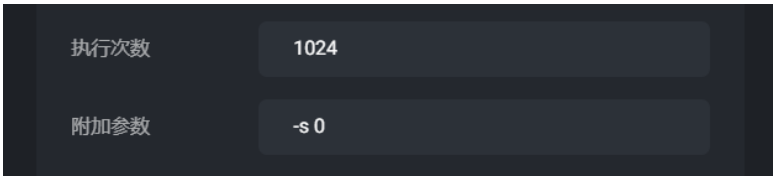
- CloudBaiduSim2Water
- CloudBaiduSim2Earth
- CloudBaiduSim2Thunder
- CloudBaiduSim2Wind
- CloudBaiduSim2Heaven
- CloudBaiduSim2Lake
- CloudAerAtBD
- CloudIoPCAS (*VIP*)
- CloudBaiduQPUQian (*VIP*)
- CloudIonAPM (*VIP*)

3.2.3 使用参数

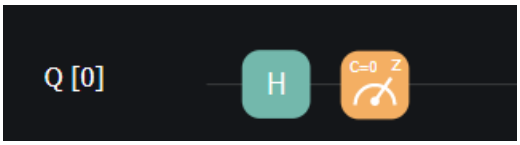
如前两小节所示，在创建量子编程环境后需要选择量子端。用户可以选择本地模拟器、云端模拟器或真机作为量子端。其中 BaiduSim2 模拟器可以通过参数 '-s' 指定随机参数 seed。使用方式如下:

```
env.backend(BackendName.LocalBaiduSim2, '-s␣0') # Local BaiduSim2
env.backend(BackendName.CloudBaiduSim2Water, '-s␣0') # Cloud BaiduSim2
```

参数范围为 [0, 2147483647]。PyOnline 和 YunIDE 使用模拟器参数的方式和 QCompute 相同，QComposer 使用模拟器参数的方法为: 点击 QComposer 界面的【设置】按钮，在【附加参数】栏输入 -s 0:



设置【运行目标】为 CloudBaiduSim2Water 后，拖动门图标创建如下电路:



运行数次，观察每次的结果，将发现它们完全一致。

注意: 固定 seed 仅能确保同一模拟器在同一个版本下计算电路时的随机行为一致。某些模拟器或版本之间的结果可能一致，但不保证它们能够一致。这是因为不同模拟器或不同版本之间的随机产生方法可能不同，而导致 state、probs、counts 结果差异，但这些结果都是符合模拟器规则的。

除了 seed 参数外，用户在提交电路任务时还可以指定执行次数参数（shots），表示重复执行并测量电路的次数。使用方式如下:

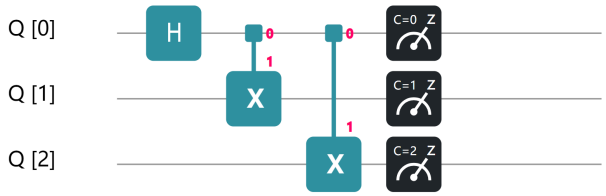
```
env.commit(shots=1024)
```

通过如上语句，设置 shots 为 1024。

第四章　代码实战

这些示例代码除子程序专项小节外均已收录到 QCompute / PyOnline / YunIDE 中，用户可在这些前端平台使用。

4.1　示例：构建 GHZ 态



本例使用 H 与 CX 门生成 GHZ 态。

- 引入所需要的包

```
from pprint import pprint
import sys
sys.path.append('../..')
from QCompute import *
```

- 输入 Token

```
Define.hubToken = "填入␣Token"
```

- 创建量子编程环境

```
env = QEnv()
```

- 选择 CloudBaiduSim2Water 作为量子端

```
env.backend(BackendName.CloudBaiduSim2Water)
```

- 编写量子电路

```
q = env.Q.createList(3)
H(q[0])
CX(q[0], q[1])
CX(q[0], q[2])
MeasureZ(*env.Q.toListPair())
```

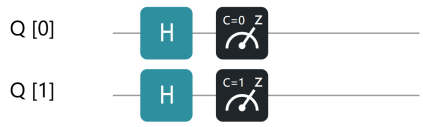
- 提交并查看结果

```
taskResult = env.commit(1024, fetchMeasure=True)
pprint(taskResult)
```

- 任务结果

```
{"000": 556, "111": 468}
```

4.2 示例: 混合语言计算



本例展示语言混合编程，恰当使用 Python 将极大提高用户在量子计算实验中的编程效率。

- 引入所需要的包

```
from pprint import pprint
import sys
sys.path.append('../..')
from QCompute import *
```

- 输入 Token

```
Define.hubToken = "填入␣Token"
```

- 创建量子编程环境

```
env = QEnv()
```

- 选择 CloudBaiduSim2Water 作为量子端

```
env.backend(BackendName.CloudBaiduSim2Water)
```

- 编写量子电路

```
TotalNumQReg = 2                        # 量子位个数
q = env.Q.createList(TotalNumQReg) # 按照量子位数定义量子寄
    存器 q

for index in range(TotalNumQReg):  # 按照量子位数逐位添加 H
    门
    H(q[index])
MeasureZ(*env.Q.toListPair())
```

- 提交并查看结果

```
taskResult = env.commit(1024, fetchMeasure=True)
pprint(taskResult)
```

- 任务结果

```
{'10': 245, '01': 252, '00': 250, '11': 253}
```

4.3 示例: 交互式计算

本例展示量子程序与 Python 程序交互，并多次提交电路到远程计算。

- 引入所需要的包

```
from pprint import pprint
import sys
sys.path.append('../..')
from QCompute import *
```

- 输入 Token

```
Define.hubToken = "填入␣Token"
```

- 设置 uValue 初始值

```
uValue = 1
```

- 用 Python 写一个循环体，循环次数为 15

```
for _ in range(15):
print("uValue␣is␣:", uValue)
```

- **在循环体内**，创建量子编程环境

```
env = QEnv()
```

- 选择 CloudBaiduSim2Water 作为量子端

```
env.backend(BackendName.CloudBaiduSim2Water)
```

- 编写量子电路

```
u = RX(uValue)
u(env.Q[0])
MeasureZ(*env.Q.toListPair())
```

- 提交电路

```
taskResult = env.commit(1024, fetchMeasure=True)
```

- 用 Python 写一个判断语句，确定 '0' 的个数小于 5 时结束循环，否则参数 `uValue * 2`

```
CountsDict = taskResult['counts']
if CountsDict.get('0', 0) > 5:
    uValue = uValue * 2
else:
    print("When␣the␣parameter␣is␣%d,␣0␣is␣eliminated." %
       uValue)
    break
```

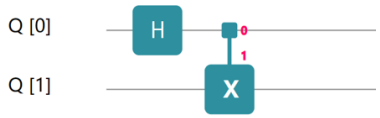
- 任务结果

```
...
When the parameter is 512, 0 is eliminated.
```

4.4 示例：量子超密编码

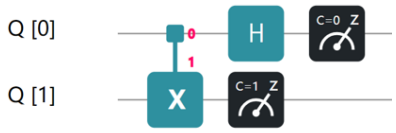
Bennett & Wiesner 于 1992 年发明量子超密编码 [1] 是通过发送一个量子比特来传输两个经典比特信息的一种量子通信协议。简单来说，量子超密编码利用量子纠缠的特性拓展了信道容量。为了说明这一通信方式，我们假设有 Alice 和 Bob 两个用户，并且他们共享一对纠缠的量子比特，且这对量子比特处于最大纠缠态（贝尔态）。

$$|\Phi^+\rangle = (|0\rangle_A \otimes |0\rangle_B + |1\rangle_A \otimes |1\rangle_B)/\sqrt{2}.$$

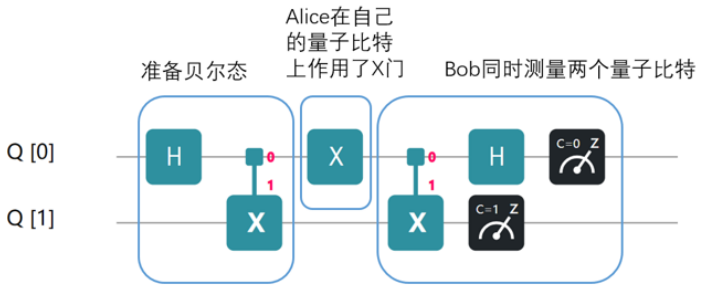


将两个量子比特初始化成量子态 $|0\rangle_A \otimes |0\rangle_B$ 并运行上图中的电路，即可制备出一个贝尔态。然后两个用户各取贝尔态中的一个量子比特，我们假设 Alice 拿到的是上面的量子比特 q_0，Bob 拿到的则是下面的 q_1。现在假设 Alice 和 Bob 分隔很远，但 Alice 想给 Bob 传输两个经典比特的信息，即 $\{00, 01, 10, 11\}$ 中的某一元素。那么，Alice 首先需要对量子比特 q_0 进行相应的操作，然后将其发送给 Bob。Bob 收到量子比特 q_0 后，对两个量子比特进行测量，便能解码出 Alice 想要传输的经典信息。

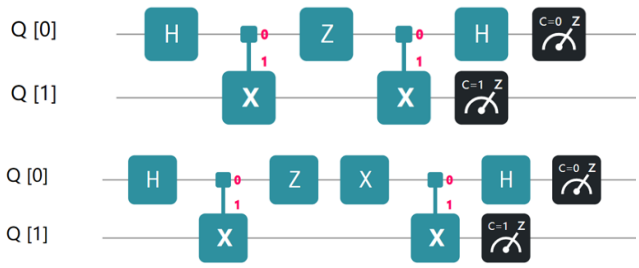
如果 Alice 想传输 00 给远处的 Bob, 那么 Alice 只需要把自己提前分到的量子比特 q_0 直接发送给 Bob。当 Bob 收到 Alice 发送的量子比特后，按顺序将两个量子比特排好位置，施加如下的电路，得到的测量结果就是 $|00\rangle$, 也就是 Alice 试图发送的信息:



如果 Alice 想发送 01 怎么办？很简单，她只需要先在自己的 qubit 上作用 X 门，然后再把自己的 qubit 发送给 Bob。Bob 收到 Alice 的 qubit，同样是将两个 qubit 排好序（Q[0] 为 Alice 的 qubit），然后做用上图的电路，测得的结果恰好就是 $|01\rangle$, 完整的过程如下:



如果 Alice 想发送 10，那么她需要先在自己的量子比特上作用 Z 门，然后再把自己的量子比特发送给 Bob。如果 Alice 想传输 11，那么她需要先在自己的量子比特上作用 Z 还有 X 门，然后再把自己的量子比特发送给 Bob。关于发送 10 和 11 的完整示意图分别如下:



这便是量子超密编码协议，本例我们展示其在 QCompute 上的实现:

- 引入所需要的包

```
from pprint import pprint
import sys
sys.path.append('../..')
from QCompute import *
```

- 设置 message 值，表示 Alice 将要传给 Bob 的信息，用户可自由选择'00', '01', '10' 和'11'

```
message = '11'
```

- 创建量子编程环境

```
env = QEnv()
```

- 选择 CloudBaiduSim2Water 作为量子端

```
env.backend(BackendName.CloudBaiduSim2Water)
```

- 编写量子电路

```
H(q[0])
CX(q[0], q[1])

if message == '01':
    X(q[0])
elif message == '10':
    Z(q[0])
elif message == '11':
    Z(q[0])
    X(q[0])

CX(q[0], q[1])
H(q[0])
MeasureZ(*env.Q.toListPair())
```

- 提交并查看结果

```
taskResult = env.commit(1024, fetchMeasure=True)
pprint(taskResult)
```

- 任务结果

```
{'11': 1024}
```

4.5 示例：变分量子基态求解器 VQE

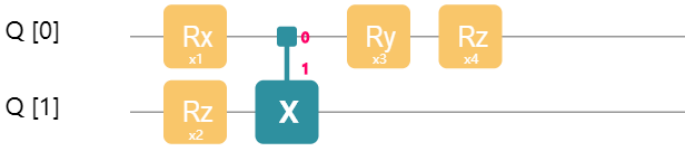
考虑这样一个数学问题：给定一个埃尔米特矩阵 H，如何找到它的最小特征值？这一问题在物理和量子化学中有直接的应用。对于一个我们感兴趣的原子级别的系统，存在着描述该系统物理性质和行为的哈密顿量，这个哈密顿量正是一个埃尔米特矩阵。通过求解最小特征值及其特征向量，我们可以找到这个量子系统的基态。变分量子基态求解器 (VQE) 就是这样一个量子算法，其原理并不复杂。假设哈密顿量 H 是一个埃尔米特矩阵，其最小特征值是 λ_{min}，那么对于任意一个单位向量 $|\phi\rangle$，我们有

$$\langle\phi|H|\phi\rangle \geq \lambda_{min}$$

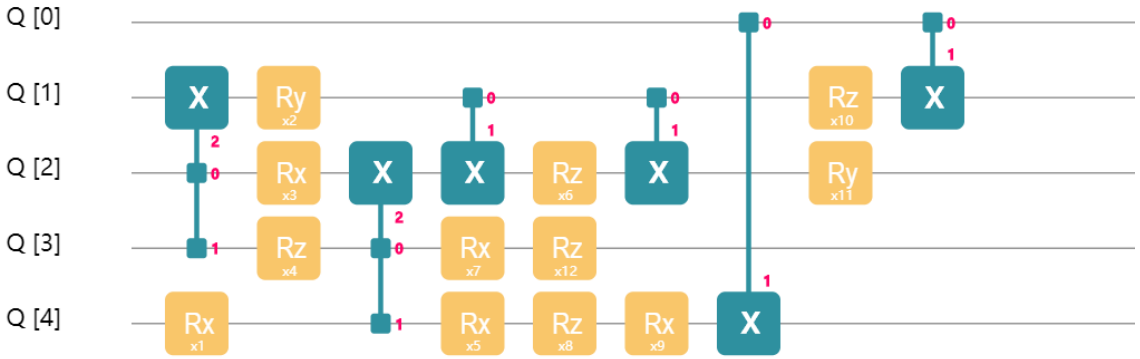
VQE 算法的核心正是通过不断调节单位向量 $|\phi\rangle$，使得 $\langle\phi|H|\phi\rangle$ 越来越接近 λ_{min}。接下来让我们了解一下 VQE 的具体实现方法，包括 $|\phi\rangle$ 的调节和 $\langle\phi|H|\phi\rangle$ 的计算。

1. **参数化量子电路**

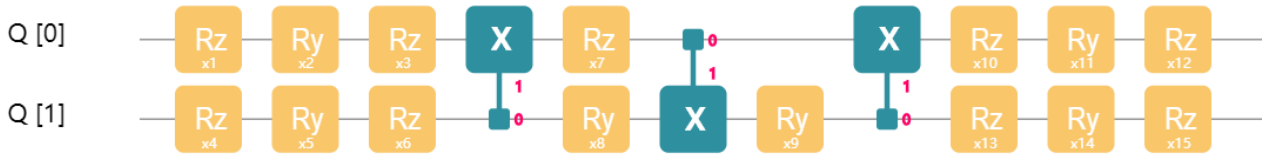
   为了调节一个单位向量 $|\phi\rangle$，我们需要建立一个参数化的量子电路。



   上图展示了一个适用于 2 个量子比特的参数化电路，这个电路由 4 个参数 (旋转门角度 x_i) 控制，在给定一个输入量子态的情况下，通过调节这 4 个参数，我们可以改变电路输出的量子态。当然，我们也可以构造更复杂的参数化电路。



   上图是一个适用于 5 个量子比特且含有 12 个参数的电路，同样地，通过调节这 12 个参数，我们可以得到不同的输出量子态。细心的读者可能会问:" 在给定一个输入量子态时，一个参数化电路能生成任意的一个量子态吗？" 这取决于这个参数化电路的构造。对于只有两个量子比特的情形，我们可以构造出一个如下图所示的含 15 个参数的量子电路。



通过调节电路中的这 15 个参数，我们可以输出两个量子比特的任意一个量子态。对于任意多个的量子比特，构造一个泛化能力很强的参数化电路现今依旧是一个开放问题。从实践上来说，只要求参数化电路输出的态 $|\phi\rangle$ 很接近真实的特征量子态 $|\phi_{\lambda_{min}}\rangle$，我们就能近似求得 $\lambda \approx \lambda_{min}$。
现在我们知道了如何利用参数化电路调节量子态 $|\phi\rangle$，下面我们介绍怎样利用量子电路计算 $\langle\phi|H|\phi\rangle$，它在物理学中被称为哈密顿量 H 在量子态 $|\phi\rangle$ 下的期望值。

2. **泡利测量**

我们通常把 H 分解成多个结构更为简单的局部哈密顿量 H_i 的线性组合 [2]:

$$H = \sum_i c_i H_i$$

复数 c_i 表示每一项的系数。其中，每一个局部哈密顿量 H_i 的期望值 $\langle\phi|H_i|\phi\rangle$ 更容易计算，最后把所有计算结果线性组合就行了。

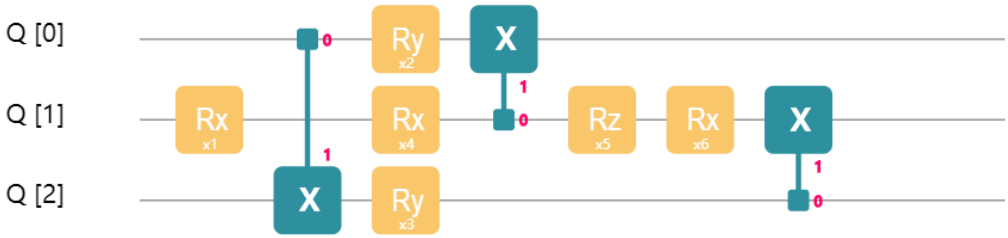
$$\langle\phi|H|\phi\rangle = \sum_i c_i \langle\phi|H_i|\phi\rangle$$

一般来说，H 可以被分解成 $O(poly(n))$ 项 H_i ，其中 n 是系统中量子比特的个数。每一项 H_i 都是泡利矩阵的张量积，例如 $\sigma_x \otimes I \otimes \sigma_z \otimes \cdots \otimes \sigma_y$。严格来说，

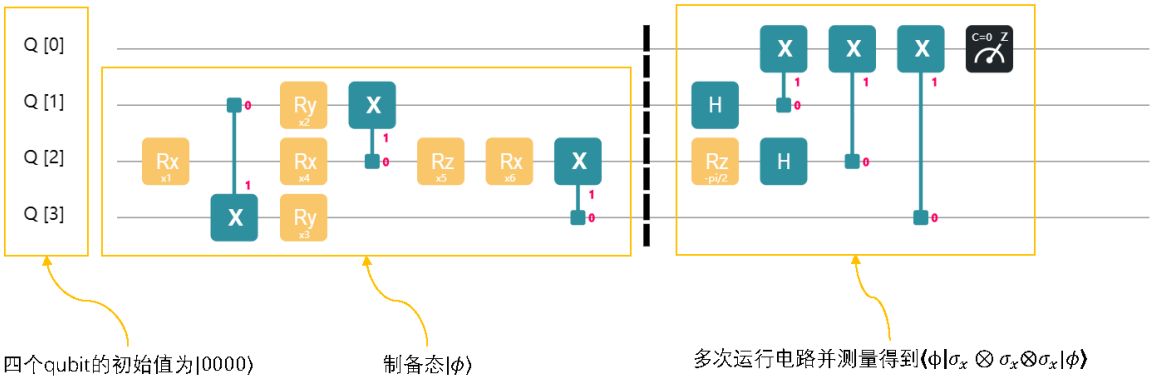
$$H_i \in \{I, \sigma_x, \sigma_y, \sigma_z\}^{\otimes n}$$

其中 I 是单位矩阵，$\sigma_x, \sigma_y, \sigma_z$ 则是 Pauli 矩阵。理论上，任何一个 $2^n \times 2^n$ 的埃尔米特矩阵 H 都可以被这样分解，也就是说，$\{I, \sigma_x, \sigma_y, \sigma_z\}^{\otimes n}$ 是一组埃尔米特矩阵基。既然得到 $\langle\phi|H|\phi\rangle$ 需要计算每一项 $\langle\phi|H_i|\phi\rangle$，那我们如何计算现在我们需要完成最后一个拼图，也就是如何计算 $\langle\phi|H_i|\phi\rangle$ 呢？一种被称为 ** 泡利测量 ** 的技术可以帮助我们解决这个问题。我们接下来看一个简单的例子。

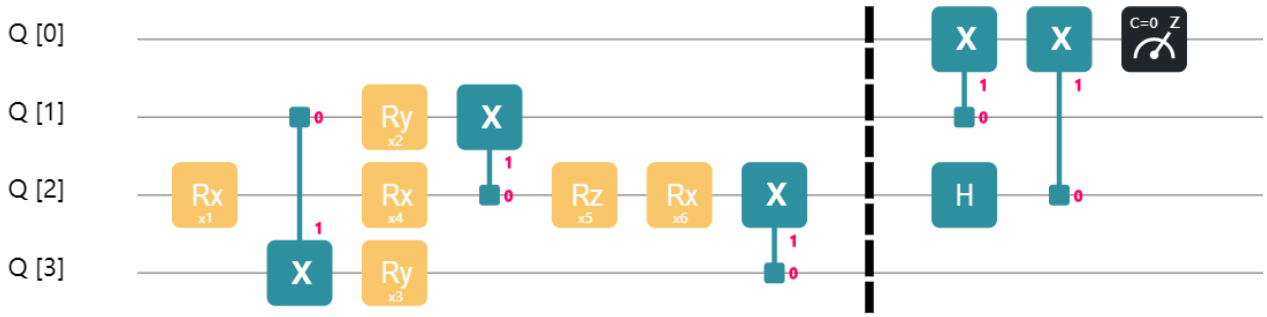
首先利用如下的参数化电路制备出 3 个量子比特的一个量子态 $|\phi\rangle$。



假设我们想得到 $\langle\phi|\sigma_x \otimes \sigma_y \otimes \sigma_z|\phi\rangle$，那么只需要搭建下图中的电路，反复进行测量。



虽然 $|\phi\rangle$ 是 3 个量子比特的量子态，但我们需要一个额外的辅助量子比特 q_0 帮助我们计算 $\langle\phi|\sigma_x \otimes \sigma_y \otimes \sigma_z|\phi\rangle$。注意到新增加的第二部分电路，它含有阿达玛门 H (** 注意 **: 这里的 H 指阿达玛门，而不是前面讨论的哈密顿量) 和 $R_z(-\pi/2)$ 门，这一部分电路是为 $\sigma_x \otimes \sigma_y \otimes \sigma_z$ 量身打造的。不同的 H_i 会对应不同的测量电路，我们下面会详细介绍。多次测量辅助量子比特后，统计出测量结果为 0 和为 1 的概率，用测量得到 0 的概率减去测量得到 1 的概率，得到的数值便是 $\langle\phi|\sigma_x \otimes \sigma_y \otimes \sigma_z|\phi\rangle$ 的近似值。测量次数越多，计算的结果就越精确。我们说到不同的 H_i 会对应不同的电路构造，现在来看一下当 $H_i = \sigma_z \otimes \sigma_x \otimes I$ 的时候，测量电路会变成什么样。

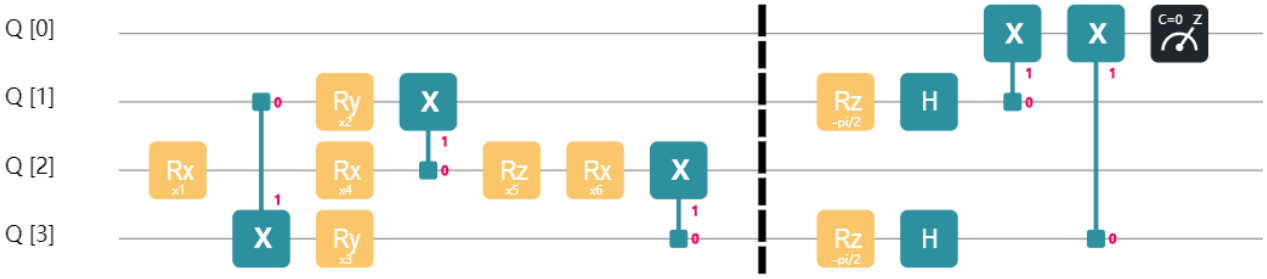


注意到只有电路的第二部分变化了，电路其它的部分则保持不变。我们仍然是让这个电路在初始态 $|0000\rangle$ 上跑好多好多次，统计出 q_0 测得 0 的概率减去测得 1 的概率，得到的数值便是 $\langle\phi|\sigma_z \otimes \sigma_x \otimes I|\phi\rangle$ 的近似值。其实这里有个重要的规律:

- σ_x 对应构造阿达玛门 $H + CNOT$ 测量
- σ_y 对应构造 $R_z(-\pi/2)$ 门 + 阿达玛门 $H + CNOT$ 测量
- σ_z 对应构造 $CNOT$ 测量
- I 单位阵对应什么也不加

回顾一下 $\sigma_x \otimes \sigma_y \otimes \sigma_z$ 所对应的电路，你发现了吗？
q_1 添加的是阿达玛门和 $CNOT$, 对应的是 $\sigma_x \otimes \sigma_y \otimes \sigma_z$ 的第一项 σ_x; q_2 添加的是 $R_z(-\pi/2)$ + 阿达玛门 + $CNOT$ ，对应的是 $\sigma_x \otimes \sigma_y \otimes \sigma_z$ 的第二项 σ_y; q_3 添加的是 $CNOT$, 对应的是 $\sigma_x \otimes \sigma_y \otimes \sigma_z$ 的第三项 σ_z。我猜你应该看出了些苗头，让我们再研究一下 $\sigma_z \otimes \sigma_x \otimes I$ 对应的电路。

q_1 添加的是 $CNOT$, 对应的是 $\sigma_z \otimes \sigma_x \otimes I$ 的第一项 σ_z; q_2 添加的是阿达玛门 + $CNOT$，对应的是 $\sigma_z \otimes \sigma_x \otimes I$ 的第二项 σ_x; q_3 什么也没加, 对应的是 $\sigma_z \otimes \sigma_x \otimes I$ 的第三项 I。让我考考你，测量 $\langle\phi|\sigma_y \otimes I \otimes \sigma_y|\phi\rangle$ 的电路会长什么样？答案见下图:



这样的话，我们已经可以得到 $H = 0.5 * \sigma_x \otimes \sigma_y \otimes \sigma_z + 0.2 * \sigma_z \otimes \sigma_x \otimes I + 0.8 * \sigma_y \otimes I \otimes \sigma_y$ 关于 $|\phi\rangle$ 的期望值 $\langle\phi|H|\phi\rangle$ 了，只要我们让前三个设计的电路每个都独立运行好多好多次，统计并计算每一

项的结果，结合系数进行线性相加就好了。我知道，第一次看起来会很复杂。但是不用怕，前面的部分多读几遍，原理其实并不深奥。

3. **梯度下降**

当你差不多弄懂了前面的细节，我们继续前进。要注意到，我们最后得到的数值其实是 $\langle\phi|H|\phi\rangle$ 的一个近似值，含有一定的统计误差。当你的测量次数越多，误差越小，你得到的结果会越精确。其次，当你变化电路中的六个参数，你会得到新的 $|\phi\rangle$。同理你会得到崭新的 $\langle\phi|H|\phi\rangle$。咦？这不就是一个多元函数？是的，在几乎所有关于 VQE 的学术论文中，学者们都会把 $\langle\phi|H|\phi\rangle$ 看成一个多元函数

$$\begin{aligned} L(\theta_1,\theta_2,\cdots,\theta_m) &= \langle\phi(\theta_1,\theta_2,\cdots,\theta_m)|H|\phi(\theta_1,\theta_2,\cdots,\theta_m)\rangle \\ &= \langle 00\cdots0|U^\dagger(\theta_1,\theta_2,\cdots,\theta_m)HU(\theta_1,\theta_2,\cdots,\theta_m)|00\cdots0\rangle \end{aligned}$$

其中 $U(\theta_1,\theta_2,\cdots,\theta_m)$ 是含参数电路所代表的矩阵。如果你把它看成多元函数，那么寻找 H 的最小特征值问题就转化成了寻找多元函数 $L(\theta_1,\theta_2,\cdots\theta_n)$ 的最小值问题。让我们梳理一下思路，我们想要找到埃尔米特矩阵 H 的最小特征值 λ_{min}，首先构造一个含参数的电路并选定一组参数，然后把 H 分解成 Pauli 矩阵张量积的线性组合并参考之前的规则在我们的参数化电路后面接入对应的测量电路，测量多次后计算得到每一个 $\langle\phi|H_i|\phi\rangle$ 的近似值并按照如下公式进行线性相加

$$L(\theta_1,\theta_2,\cdots\theta_m) = \langle\phi|H|\phi\rangle = \sum_i c_i\langle\phi|H_i|\phi\rangle$$

由此我们得到了 $\langle\phi|H|\phi\rangle$ 的近似值。这给了我们一个想法，我们可以计算 $L(\theta_1,\theta_2,\cdots\theta_n)$ 的梯度。利用经典的梯度下降，我们就可以更新参数。迭代多次以后，我们就能找到 $L(\theta_1,\theta_2,\cdots\theta_n)$ 的最小值了！选择一个足够小的 ϵ，利用

$$\frac{\partial L}{\partial\theta_i} \approx \frac{L(\cdots,\theta_i+\epsilon,\cdots)-L(\cdots,\theta_i,\cdots)}{\epsilon}$$

我们就可以得到完整的梯度，这种方法被称为有限差分方法。由于 VQE 统计误差的存在，有限差分有严重的精度缺陷。因此我们倾向于使用解析梯度的方法获得 $L(\theta_1,\theta_2,\cdots\theta_n)$ 的梯度

$$\frac{\partial L}{\partial\theta_i} = \frac{L(\cdots,\theta_i+\pi/2,\cdots)-L(\cdots,\theta_i-\pi/2,\cdots)}{2}$$

注意，我们算出的是精确的梯度而不是有限差分的近似梯度！解析梯度精度不会被严重干扰，因此成了 VQE 进行梯度计算的首选工具。下面就让我们在量易伏上实战演练一下！

4. **在量易伏上实现 VQE**

- 首先调用需要的包

```
from copy import copy
import multiprocessing as mp
import pickle
import random
from functools import reduce
from os import path
import numpy as np
import scipy
import scipy.linalg
```

```
from matplotlib import pyplot as plt
import sys
sys.path.append('../../..')
from QCompute import *
```

- 然后我们进行各种参数及超参数的设置

```
# hyper-parameter setting
shots = 1024
# n must be larger than or equal to 2;
# n is the size of our quantum system
n = 4
assert n >= 2
L = 2  # L is the number of layers
iteration_num = 20
# That's the number of parallel experiments we will run;
experiment_num = 4
# it indicates the number of processes we will use.
# Don't stress your computer too much.
learning_rate = 0.3
delta = np.pi / 2  # calculate analytical derivative
# This number will determine what the final Hamiltonian is.
# It is also used to make sure Mac and Windows behave the
# same using multiprocessing module.
SEED = 36
K = 3  # k is the number of local hamiltonian in H
# N is the number of parameters needed for the circuit
N = 3 * n * L
random.seed(SEED)
```

- 定义可以随机生成哈密顿量的函数

```
def random_pauli_generator(l):
    """
    The following functions are used to generate random
       Hamiltonian
    """
    s = ''
    for i in range(l):
        s = s + random.choice(['i', 'x', 'y', 'z'])
    return s


def random_H_generator(n, k):
    """
    n is the number of qubits,
    k is the number of local hamiltonian in H
    """
```

```
    H = []
    for i in range(k):
        H.append([random.random(), random_pauli_generator(n)
            ])
    return H


Hamiltonian = random_H_generator(n, K)  # our hamiltonian H
```

- 借用 Paddle Quantum 中计算多个矩阵张量积的函数

```
def NKron(AMatrix, BMatrix, *args):
    """
    Recursively execute kron n times.
    This function at least has two matrices.
    :param AMatrix: First matrix
    :param BMatrix: Second matrix
    :param args: If have more matrix,
    they are delivered by this matrix
    :return: The result of tensor product.
    """
    return reduce(
        lambda result, index: np.kron(result, index),
        args,
        np.kron(AMatrix, BMatrix), )
```

- 定义计算一个哈密顿量最小特征值的函数

```
def ground_energy(Ha):
        """
        It returns the ground energy of hamiltonian Ha, which
        looks like [[12, 'xyiz'], [21, 'zzxz'], [10, 'iixy
            ']].
        """
        # It is a local function
        def my_f(s):
            s = s.lower()
            I = np.eye(2) + 0j
            X = np.array([[0, 1], [1, 0]]) + 0j
            Y = np.array([[0, -1j], [1j, 0]])
            Z = np.array([[1, 0], [0, -1]]) + 0j
            if s == 'x':
                return X
            elif s == 'y':
                return Y
            elif s == 'z':
                return Z
            else:
```

```
            return I

        # It is a local function
        def my_g(s_string):
            H = []
            for ele in s_string:
                H.append(my_f(ele))
            return NKron(*H)
        sum = 0
        for ele in Ha:
            sum += ele[0] * my_g(ele[1])
        eigen_vector = np.sort(scipy.linalg.eig(sum)[0])
        return eigen_vector[0].real
```

- 这是一个帮助我们对实验结果进行可视化的函数

```
    def eigen_plot(eigenv_list, actual_eigenv):
        """
        This is the plot function of actual loss over iterations.
        """
        for ele in eigenv_list:
            plt.plot(list(range(1, len(ele) + 1)), ele, linewidth
               =4)
        plt.axhline(y=actual_eigenv, color='black', linestyle='-.
           ')
        plt.xlabel('iteration')
        plt.ylabel('loss')
        plt.title('Actual␣Loss␣Over␣Iteration')
        plt.show()
```

- 接下来的这个函数用来处理实验的测量结果，近似估计出那个多余的 qubit 测量得到 0 的概率减去得到 1 的概率

```
    def prob_calc(data_dic):
        """
        Measure the first (ancilla) qubit. Returns the value
        of 'the probability of getting 0' minus
        'the probability of getting 1'.
        """
        sum_0 = 0
        sum_1 = 0
        for key, value in data_dic.items():
            if int(list(key)[-1], 16) % 2 == 0:
                sum_0 += value
            else:
                sum_1 += value
        return (sum_0 - sum_1) / shots
```

- 下面的函数用帮助生成参数化电路

```
def add_block(q, loc, para):
    """
    Add a RzRyRz gate block. Each block has 3 parameters.
    """
    RZ(para[0])(q[loc])
    RY(para[1])(q[loc])
    RZ(para[2])(q[loc])


def add_layer(para, q):
    """
    add a layer, each layer has 3*n parameters.
    para is a 2-D numpy array
    """
    for i in range(1, n + 1):
        add_block(q, i, para[i-1])
    for i in range(1, n):
        CX(q[i], q[i+1])
    CX(q[n], q[1])
```

- 然后我们根据线性分解后的哈密顿量，针对性的设计相应的测量电路

```
def self_defined_circuit(para, hamiltonian):
    """
    H is a list, for example, if
    H = 12*X*Y*I*Z + 21*Z*Z*X*Z + 10* I*I*X*Y,
    then parameter hamiltonian is
    [[12, 'xyiz'], [21, 'zzxz'], [10, 'iixy']]
    (upper case or lower case are all fine).
    It returns the expectation value of H.
    """
    env = QuantumEnvironment()
    env.backend(BackendName.LocalBaiduSim2)
    # the first qubit is ancilla
    q = [env.Q[i] for i in range(n + 1)]
    hamiltonian = [symbol.lower() for symbol in hamiltonian]
    # change 1-D numpy array to a 3-D numpy array
    high_D_para = para.reshape(L, n, 3)
    # set up our parameterized circuit
    for i in range(1, n + 1):
        H(q[i])
    # add parameterized circuit
    for i in range(L):
        add_layer(high_D_para[i], q)
    for i in range(n):
```

```
            # set up pauli measurement circuit
            if hamiltonian[i] == 'x':
                H(q[i + 1])
                CX(q[i + 1], q[0])
            elif hamiltonian[i] == 'z':
                CX(q[i + 1], q[0])
            elif hamiltonian[i] == 'y':
                RZ(-np.pi / 2)(q[i + 1])
                H(q[i + 1])
                CX(q[i + 1], q[0])
        # measurement result
        MeasureZ([env.Q[i] for i in range(n)], range(n))
        taskResult = env.commit(shots, fetchMeasure=True)
        return prob_calc(taskResult['counts'])
```

- 接下来是一个计算梯度，并更新参数的函数

```
def diff_fun(f, para):
    """
    It calculates the gradient of f on para,
    update parameters according to the gradient,
    and return the updated parameters.'para' is a np.array.
    """
    para_length = len(para)
    gradient = np.zeros(para_length)
    for i in range(para_length):
        para_copy_plus = copy(para)
        para_copy_minus = copy(para)
        para_copy_plus[i] += delta
        para_copy_minus[i] -= delta
        gradient[i] = (f(para_copy_plus) - f(para_copy_minus)
            ) / 2
    new_para = copy(para)
    res = new_para - learning_rate * gradient
    return res
```

- 计算 $\langle\phi|H|\phi\rangle$

```
def loss_fun(para):
    """
    This is the loss function.
    """
    res = sum([ele[0] *
            self_defined_circuit(para, ele[1]) for ele in
               Hamiltonian])
    return res
```

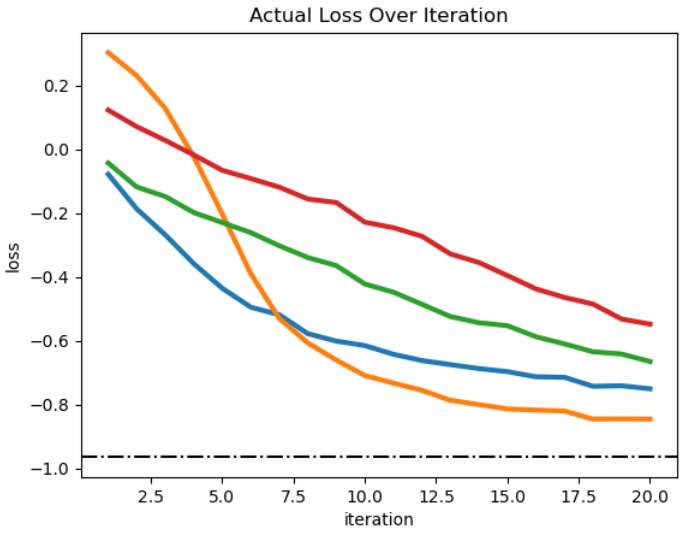
- 由于要用到 multiprocessing 这个包，我们要将进行一些封装

```
def multi_process_fun(j):
    """
    This function runs one experiment,
    parameter j indicates it is the j-th experiment.
    """
    np.random.seed()
    para = np.random.rand(N) * np.pi
    para_list = [para]
    for i in range(iteration_num):
        para_list.append(diff_fun(loss_fun, para_list[i]))
    with open(path.join(outputPath, f"para{j}.pickle"),
            "wb") as fp:
        pickle.dump(para_list, fp)
```

- 最后我们可以运行主程序了

```
def main():
    """
    main
    """
    pool = mp.Pool(experiment_num)
    pool.map(multi_process_fun, range(experiment_num))
    loss_list = []
    for _ in range(experiment_num):
        actual_loss = []
        with open(path.join(outputPath,  f"para{_}.pickle"),
                "rb") as fp:
            new_para_list = pickle.load(fp)
        for j in range(iteration_num):
            actual_loss.append(loss_fun(new_para_list[j]))
        loss_list.append(actual_loss)
    eigen_plot(loss_list, ground_energy(Hamiltonian))

if __name__ == '__main__':
    main()
```

- 下面这张图是由上述代码跑出的实验结果，每一根线都是一个不同的 process 独立进行的 20 次梯度下降优化，可以看到效果还不错。



4.6 示例：量子子程序

子程序 (Subroutines) 功能在 QComposer 中扮演着函数的角色，它使得量子编程中的参数和量子寄存器使用更灵活。子程序的使用将使用户的量子编程更加得心应手。

4.6.1 QComposer 上实现子程序

1. 功能区说明

2. 用例说明

在分解 $15 = 3 \times 5$ 这一任务中，Shor 算法需要计算一个整数模 15 的乘法阶。我们选取这个整数为 2，将 2 的各个幂用四个两比特量子态来编码：

$$|1\rangle \to |00\rangle,\ |2\rangle \to |01\rangle,\ |4\rangle \to |10\rangle,\ |8\rangle \to |11\rangle.$$

使用如下量子相位估计的电路，当观测到量子态 $|01\rangle$ 或 $|11\rangle$ 时，可以近似认为 2 模 15 的乘法阶为 4（来自 $\frac{(01)_2}{2^2}$ 和 $\frac{(11)_2}{2^2}$ 的分母）。

将阶 4 代入下述公式即可得到 15 的分解：

$$15 = \gcd(2^{4/2} - 1, 15) \times \gcd(2^{4/2} + 1, 15) = 3 \times 5.$$

更多信息可以参考 QCompute SDK 或者量易简。

3. 定义子程序 CMM

CMM 是一个 3 比特受控模乘门，由 CX 门和 CCX 门组成。

- 在量子门和子程序选中 Subroutines，点击 Add 自动创建一个名为 nG0 的空白子程序；
- 点击 nG0 图标，点击出现在图标左上角的编辑按钮，进入子程序编辑界面。点击 Operation nG0 旁边的编辑按钮，将子程序重命名为 CMM；
- 点击加号按钮，添加两个量子寄存器 Q[1] 和 Q[2]；
- 在量子门和子程序选中 Operations，从中向子程序电路图拖一个 CCX 门，单击 CCX 门，点击出现在左上角的编辑按钮，进入量子门编辑界面，将 Q[1] 上的连线拖动到 Q[2]，退出量子门编辑界面；
- 从量子门和子程序向子程序电路图拖一个 CX 门，单击 CX 门，点击出现在左上角的编辑按钮，进入量子门编辑界面，将 Q[1] 上的连线拖动到 Q[2]，退出量子门编辑界面；
- 退出子程序编辑界面即可看到 Subroutines 下多了一个名为 CMM 的子程序。

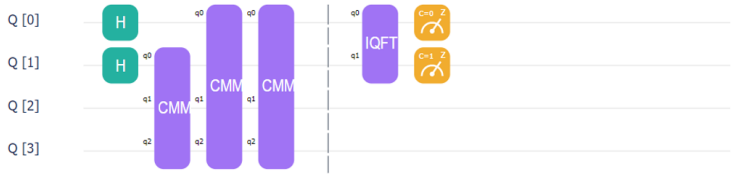
4. 定义子程序 IQFT

IQFT 完成 2 比特逆量子傅里叶变换，由 SWAP 门，H 门和 CU 门组成。

- 在量子门和子程序选中 Subroutines，点击 Add 自动创建一个名为 nG0 的空白子程序；
- 点击 nG0 图标，点击出现在图标左上角的编辑按钮，进入子程序编辑界面。点击 Operation nG0 旁边的编辑按钮，将子程序重命名为 IQFT；
- 点击加号按钮，添加一个量子寄存器 Q[1]；
- 在量子门和子程序选中 Operations，从中向子程序电路图拖一个 SWAP 门；
- 从量子门和子程序向子程序电路图拖一个 H 门；
- 从量子门和子程序向子程序电路图拖一个 C‘ 门，单击 CU 门，点击出现在左上角的编辑按钮，进入量子门编辑界面，将 Q[0] 上的连线拖动到 Q[1]，在 theta 框输入 0，在 lambda 框输入 -pi/2，退出量子门编辑界面；
- 从量子门和子程序向子程序电路图拖一个 H 门；
- 退出子程序编辑界面即可看到 Subroutines 下多了一个名为 IQFT 的子程序。

5. 调用子程序

调用子程序的方式和使用量子门的方式是一样的，按照需要将其拖动到电路中即完成调用。



用户可尝试按照上图所示，自行完成电路创建，请注意子程序的量子寄存器顺序应当和上图保持一致，q0 表示子程序内的 Q[0]，以此类推。

完整的电路 QASM 代码如下:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[4];
creg c[4];
gate nG0(param0) qb0
{
}

gate CMM qb0,qb1,qb2
{
  ccx qb0, qb2, qb1;
  cx qb0, qb2;
}

gate IQFT qb0,qb1
{
  swap qb0, qb1;
  h qb1;
  cu(0, 0, -pi/2) qb1, qb0;
  h qb0;
}

h q[0];
h q[1];
CMM  q[1], q[2], q[3];
CMM  q[0], q[2], q[3];
CMM  q[0], q[2], q[3];
barrier  q[0], q[1], q[2], q[3];
IQFT  q[0], q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
```

4.6.2 QCompute 上实现子程序

1. 定义子程序 CMM

   CMM 是一个 3 比特受控模乘门，由 CX 门和 CCX 门组成。

```
CMMEnv = QEnv()
CCX(CMMEnv.Q[0], CMMEnv.Q[1], CMMEnv.Q[2])
CX(CMMEnv.Q[0], CMMEnv.Q[2])
CMM = CMMEnv.convertToProcedure('CMM', env)
```

2. 定义子程序 IQFT

   IQFT 完成 2 比特逆量子傅里叶变换，由 SWAP 门，H 门和 CU 门组成。

```
IQFTEnv = QEnv()
SWAP(CMMEnv.Q[0], CMMEnv.Q[1])
H(CMMEnv.Q[1])
CU(0, 0, -np.pi/2)(CMMEnv.Q[1], CMMEnv.Q[0])
H(CMMEnv.Q[0])
IQFT = IQFTEnv.convertToProcedure('IQFT', env)
```

3. 调用子程序

   调用子程序的方式和使用量子门的方式是一样的。

完整的电路 Python 代码如下:

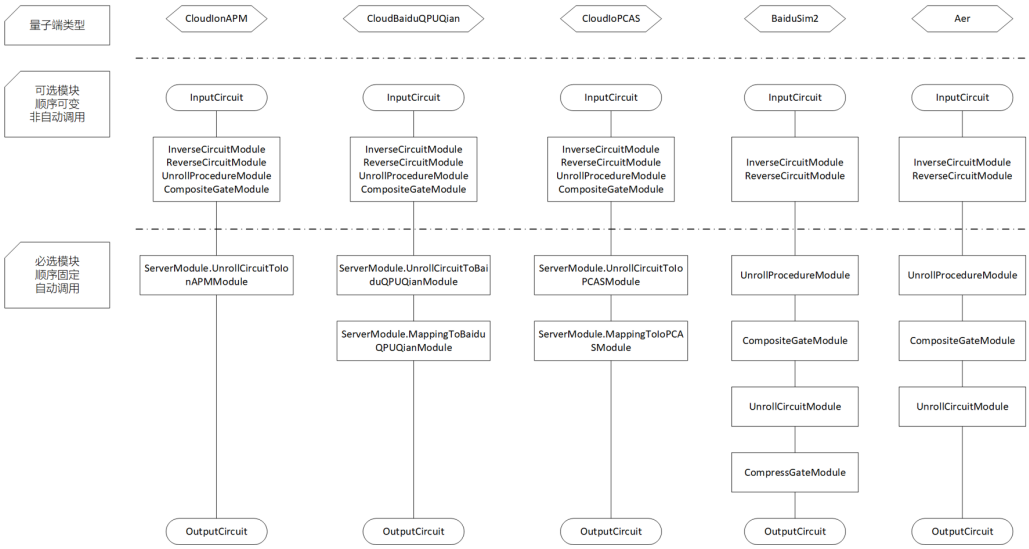
```
from QCompute import *
from pprint import pprint
from QCompute.Define import Settings
Settings.outputInfo = False
import numpy as np

matchSdkVersion('Python 3.0.0')

# Create environment
env = QEnv()
# Choose backend Baidu local simulator
env.backend(BackendName.LocalBaiduSim2)

# Initialize the three-qubit circuit
q = env.Q.createList(3)

# Define subroutine CMM
CMMEnv = QEnv()
CCX(CMMEnv.Q[0], CMMEnv.Q[1], CMMEnv.Q[2])
CX(CMMEnv.Q[0], CMMEnv.Q[2])
CMM = CMMEnv.convertToProcedure('CMM', env)

# Define subroutine IQFT
IQFTEnv = QEnv()
SWAP(CMMEnv.Q[0], CMMEnv.Q[1])
H(CMMEnv.Q[1])
```

```
CU(0, 0, -np.pi/2)(CMMEnv.Q[1], CMMEnv.Q[0])
H(CMMEnv.Q[0])
IQFT = IQFTEnv.convertToProcedure('IQFT', env)

# Call subroutines
H(q[0])
H(q[1])
CMM(q[1], q[2], q[3])
CMM(q[0], q[2], q[3])
CMM(q[0], q[2], q[3])
Barrier(*q)
IQFT(q[0], q[1])

MeasureZ(*env.Q.toListPair())

# Commit the task with 1024 shots
taskResult = env.commit(1024, fetchMeasure=True)

pprint(taskResult)
```

4.7 示例：模块使用示例

模块使用规则如下图所示，“可选模块”需要用户使用相应语句声明调用，用户根据构建电路的需求选用“可选模块”，使用顺序不作限制。“必选模块”是一个电路中必然使用到的模块，不同的量子端类型，其“必选模块”也不同。“必选模块”将由系统自动调用，用户无须声明调用，用户可以关闭“必选模块”或 DIY 相应功能，但这可能导致电路无法运行的情况。

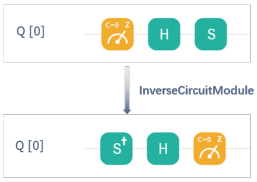
模块流程图



4.7.1 可选模块

1. InverseCircuitModule

   InverseCircuitModule 用于生成逆电路。需注意，测量为每个量子寄存器的最后一个操作，且每个量子寄存器至多进行一次测量。

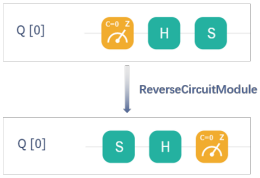



```
import sys
sys.path.append('../..')
from QCompute import *

# 创建量子环境 env，并初始化量子比特数为 1
env = QEnv()
env.backend(BackendName.LocalBaiduSim2)
q = env.Q.createList(1)

# 添加量子门
MeasureZ(*env.Q.toListPair())
```

```
H(q[0])
S(q[0])

# 调用 InverseCircuitModule
env.module(InverseCircuitModule())

# 关闭控制台绘制电路
# 关闭 UnrollCircuitModule 和 CompressGateModule
from QCompute.Define import Settings
Settings.drawCircuitControl = []
env.module(UnrollCircuitModule({'disable': True}))
env.module(CompressGateModule({'disable': True}))

# 输出电路代码
env.publish()
print(env.program)
```

2. ReverseCircuitModule

ReverseCircuitModule 用于生成反电路。需注意，测量为每个量子寄存器的最后一个操作，且每个量子寄存器至多进行一次测量。




```
import sys
sys.path.append('../..')
from QCompute import *

# 创建量子环境 env，并初始化量子比特数为 1
env = QEnv()
env.backend(BackendName.LocalBaiduSim2)
q = env.Q.createList(1)

# 添加量子门
MeasureZ(*env.Q.toListPair())
H(q[0])
S(q[0])

# 调用 ReverseCircuitModule
env.module(ReverseCircuitModule())
```

```
# 关闭控制台绘制电路
# 关闭 UnrollCircuitModule 和 CompressGateModule
from QCompute.Define import Settings
Settings.drawCircuitControl = []
env.module(UnrollCircuitModule({'disable': True}))
env.module(CompressGateModule({'disable': True}))

# 输出电路代码
env.publish()
print(env.program)
```

4.7.2 必选模块

1. UnrollProcedureModule

   UnrollProcedureModule 与量子子程序配套使用，将量子子程序内联展开到主程序中。子程序（Subroutines）功能在量易伏中扮演函数的角色，善用子程序功能将使您的量子编程更加得心应手。

   - 参数说明

     ```
     - disable
     # 关闭 UnrollProcedureModule，所有子程序不被展开而保留在输出
         电路中，这可能导致电路无法运行
     env.module(UnrollProcedureModule({'disable': True}))
     # 开启 UnrollProcedureModule，所有子程序被展开为对应的量子门
     env.module(UnrollProcedureModule({'disable': False}))
     ```

   - 声明子程序

     如下示例代码所示，首先创建量子环境 procedure1Env。其次，添加量子门 RX，设置角参数为 procedure1Env.Parameter[0]，作用在 procedure1Env.Q[0] 上。然后调用子程序 procedure0，作用在 procedure1Env.Q[1] 和 procedure1Env.Q[0] 上。最后使用 convertToProcedure 工具生成子程序，命名为 procedure1 并添加到主程序的量子环境 env 中。

     ```
     procedure1Env = QEnv()
     RX(procedure1Env.Parameter[0])(procedure1Env.Q[0])
     procedure0()(procedure1Env.Q[1], procedure1Env.Q[0])
     procedure1 = procedure1Env.convertToProcedure('procedure1',
         env)
     ```

   - 子程序转换工具

     QCompute SDK 提供快捷生成子程序逆电路和反电路的工具。如下所示，基于子程序 procedure2 生成他的逆电路和反电路。

     ```
     procedure2__inversed, _ = env.inverseProcedure('procedure2')
     procedure2__reversed, _ = env.reverseProcedure('procedure2')
     ```

   - 调用子程序

     如下使用简单的语句即可调用子程序，为区分子程序的边界，方便查看，在被调用的子程序之间添加 Barrier。

```
procedure0()(q[0], q[1])
Barrier(*q)
procedure1(6.4)(q[1], q[2])
Barrier(*q)
procedure2()(q[0], q[1])
Barrier(*q)
procedure2__inversed()(q[0], q[1])
Barrier(*q)
procedure2__reversed()(q[0], q[1])
Barrier(*q)
```

2. UnrollCircuitModule

UnrollCircuitModule 用于展开量子电路，默认展开所有门为 U 和 CX 门，使其适配仅支持 U 和 CX 门的 Sim2 模拟器。

- 支持的门
  固定门：ID, X, Y, Z, H, S, SDG, T, TDG, CX, CY, CZ, CH, SWAP, CCX, CSWAP
  旋转门：U, RX, RY, RZ, CU, CRX, CRY, CRZ

- 参数说明

  - disable

  `env.module(UnrollCircuitModule({'disable': True}))`，关闭 UnrollCircuitModule，所有门不被展开而保留在输出电路中，这可能导致电路无法运行；

  `env.module(UnrollCircuitModule({'disable': False}))`，开启 UnrollCircuitModule，所有门被展开为 U 和 CX 门。

  - errorOnUnsupported

  `env.module(UnrollCircuitModule({'errorOnUnsupported': True}))`，遇到不能处理的门会报错；

  `env.module(UnrollCircuitModule({'errorOnUnsupported': False}))`，忽略不支持的门，不支持的门不被展开而保留在输出电路中。

  - targetGates

  目标模拟器 / 真机支持的门列表（默认为 U 和 CX），这些门不会被展开而保留在输出电路中，至少包括 U 和 CX 门。如：

  `env.module(UnrollCircuitModule({'targetGates': ['U', 'CX', 'S']}))`，电路中的 U、CX 和 S 门不会被展开而保留在输出电路中。

  - sourceGates

  指定分解的门列表（默认为电路中所有门），只有被指定的门会被分解，其他门不会被展开而保留在电路中。如：

  `env.module(UnrollCircuitModule({'sourceGates': ['S']}))`，电路中的 S 门会被展开，其他门不会被展开而保留在输出电路中。

3. CompressGateModule

   CompressGateModule 用于压缩量子电路，输入任意量子电路，输出压缩后的量子电路，压缩后电路中的门个数会减少。

4. 参数说明

   - disable

     env.module(CompressGateModule({'disable': True}))，关闭 CompressGateModule，量子电路不会被压缩；

     env.module(CompressGateModule({'disable': False}))，开启 CompressGateModule，量子电路被压缩，输出电路中门的个数会减少。

4.7.3 示例: 真机相关模块

以 CloudBaiduQPUQian 为例,其他真机相关模块及其使用请参照 QCompute SDK 的 Example/Level_1。

```
from pprint import pprint
import sys
sys.path.append('../..')  # "from QCompute import *" requires this
from QCompute import *
# 设置 hubToken
# Define.hubToken = ""
# 创建量子编程环境
env = QEnv()
# 选择 CloudBaiduQPUQian 作为量子后端
env.backend(BackendName.CloudBaiduQPUQian)
# 初始化电路
q = env.Q.createList(3)

# 例 1:
# 本例说明 ServerModule.UnrollCircuitToBaiduQPUQian() 和 ServerModule.
   MappingToBaiduQPUQian() 模块的作用，这两个模块在用户使用
   CloudBaiduQPUQian 作为后端时自动调用
# ServerModule.UnrollCircuitToBaiduQPUQian 模块将 H 门和 CH 门分解为 QPU
    支持的 Rx,Ry 和 CZ 门，ServerModule.MappingToBaiduQPUQian 模块将双量
   子位门 (CZ) 中的两个逻辑量子位映射到两个耦合的物理量子位
# H(q[0])
# CH(q[0], q[1])
# CH(q[0], q[2])

# 例 2:
# 本例关闭 ServerModule.UnrollCircuitToBaiduQPUQian()，以观察其作用
# H(q[0])
# CH(q[0], q[1])
# CH(q[0], q[2])
```

```
# env.serverModule(ServerModule.UnrollCircuitToBaiduQPUQian, {"disable":
    True})

# 例 3:
# 本例关闭 ServerModule.MappingToBaiduQPUQian(), 以观察其作用
# H(q[0])
# CH(q[0], q[1])
# CH(q[0], q[2])
# env.serverModule(ServerModule.MappingToBaiduQPUQian, {"disable": True
   })
# Measure with the computational basis
MeasureZ(*env.Q.toListPair())

env.publish()
print(env.program)
# 提交任务并查看结果
taskResult = env.commit(4000, fetchMeasure=True)
pprint(taskResult)
```

第五章　API

5.1　量子门 (Gate)

量子门是量子计算——特别是量子电路的计算模型——中基本的操作一个或数个量子位的量子电路。它是量子电路的基础，类似传统逻辑门跟一般数字电路之间的关系。与多数传统逻辑门不同，量子逻辑门是可逆的。然而，传统的计算可以仅使用可逆的门表示。举例来说，可逆的 CCX/Toffoli 门可以实做所有的布尔函数。该门有一个直接等同的量子门，也因此代表量子电路可以模拟所有传统电路的操作（参考链接）。量易伏支持以下量子门:

1. 固定门: ID、X、Y、Z、H、S、SDG、T、TDG、CX、CY、CZ、CH、SWAP、CCX、CSWAP；
2. 旋转门: U (U1 / U2 / U3)、R、RX、RY、RZ、CR、CRX、CRY、CRZ；
3. 组合门: RZZ。

5.1.1　固定门 (Fixed Gate)

固定门是最基础的一类量子门，其参数是若干量子寄存器。固定门及其用法如下:

ID

```
ID(q[0])
```

X

```
X(q[0])
```

Y

```
Y(q[0])
```

Z

```
Z(q[0])
```

H

```
H(q[0])
```

S

```
S(q[0])
```

SDG

```
SDG(q[0])
```

T

```
T(q[0])
```

TDG

```
TDG(q[0])
```

CX

```
CX(q[0], q[1])
```

CY

```
CY(q[0], q[1])
```

CZ

```
CZ(q[0], q[1])
```

CH

```
CH(q[0], q[1])
```

SWAP

```
SWAP(q[0], q[1])
```

CCX

```
CCX(q[0], q[1], q[2])
```

CSWAP

```
CSWAP(q[0], q[1], q[2])
```

5.1.2 旋转门 (Rotation Gate)

旋转门是的行为受角度控制，其参数是角度与若干量子寄存器。旋转门及其用法如下:

U

```
U(angle1, angle2, angle3)(q[0])
```

RX

```
RX(angle1)(q[0])
```

RY

```
RY(angle1)(q[0])
```

RZ

```
RZ(angle1)(q[0])
```

CU

```
CU(angle1, angle2, angle3)(q[0], q[1])
```

CRX

```
CRX(angle1)(q[0], q[1])
```

CRY

```
CRY(angle1)(q[0], q[1])
```

CRZ

```
CRZ(angle1)(q[0], q[1])
```

5.1.3 组合门 (Composite Gate)

组合门是由固定门与旋转门组合构成的门电路，其参数是角度与若干量子寄存器。组合门及其用法如下:

RZZ

- 需 CompositeGate 模块配合使用:

```
# 创建电路
RZZ(angle1, angle2, angle3)(q[0], q[1])
...
# 加入处理模块
env.module(CompositeGate())
```

- 门被展开为:

```
CX(env.Q[0], env.Q[1])
U(a1, a2, a3)(env.Q[1])
CX(env.Q[0], env.Q[1])
```

5.1.4 其他

Barrier

```
Barrier(q[0], q[1], q[2])
```

MeasureZ: 测量门
参数为一组量子寄存器和传统寄存器的列表，置于量子电路的末尾。

```
MeasureZ(*env.Q.toListPair())
```

5.1.5 模块 (Module)

模块用于处理量子电路，其输入、输出均为量子电路。多个模块以串行方式使用，代码中先创建的模块先调用。在本地或服务器执行时会先按照列表中的模块依次进行处理后再进行执行和测量。模块分为 Module (不开源的模块，仅限后台执行) 和 OpenModule (开源的模块，前后台都可以执行)。

InverseCircuitModule: 生成逆电路

InverseCircuitModule 用于生成逆电路。

```
env.module(InverseCircuitModule())
```

ReverseCircuitModule: 生成反电路

ReverseCircuitModule 用于生成反电路。

```
env.module(ReverseCircuitModule())
```

CompositeGateModule: 组合门展开器

CompositeGateModule 用于展开组合门，输入任意量子电路，将其中的组合门展开。该模块与组合门，如 RZZ，配合使用。

```
env.module(CompositeGate())
```

UnrollProcedureModule: 展开量子子程序

UnrollProcedureModule 与量子子程序配套使用，将量子子程序内联展开到主程序中。

```
env.module(UnrollProcedureModule())
```

CompressGateModule: 压缩量子电路

CompressGateModule 用于压缩量子电路，输入任意量子电路，输出压缩后的量子电路，压缩后电路中的门个数会减少。

```
env.module(CompressGate())
```

UnrollCircuitModule: 所有门展开为 U 和 CX 门

UnrollCircuitModule 用于电路展开，使其适配仅支持 U 和 CX 门的 Sim2 模拟器，输入任意量子电路转换为可执行的量子电路。输入参数 errorOnUnsupported 为真时，遇到不能处理的门会报错，否则会忽略不支持的门，不支持的门不被展开而保留在输出电路中。输入参数 targetGates 为目标模拟器 / 真机支持的门列表，这些门不会被展开保而保留在输出电路中，且不会因此报错，一般情况至少包括 CX 和 U 两个基础门。有组合门时，需配合 CompositeGateModule，先展开组合门，再展开固定 / 旋转门。该模块与 Sim2 模拟器配合使用。

```
env.module(env.module(UnrollCircuitModule()))
```

5.2 量子端/模拟器 (Backend/Simulator)

量子端描述了量子电路的执行方式，分本地和远程两种，提交量子计算时应选择其中之一。量易伏设有用户分级措施: 普通用户可用本地模拟器、服务端模拟器；VIP 用户除模拟器外，还可以通过 QPU 方式使用第三方量子计算机。量易伏可用的量子端如下表所示:

类型	量子端	说明
本地	LocalBaiduSim2	使用 Python 编写的 Sim2 本地版
云端	CloudBaiduSim2Water	使用 C++ 编写的多实例 Sim2 模拟器云版
	CloudBaiduSim2Earth	使用 Python 编写的单一实例高配置 Sim2 模拟器云版
	CloudBaiduSim2Thunder	使用 C++ 编写的单一实例高配置 Sim2 模拟器云版
	CloudBaiduSim2Wind	使用 C++ 编写的单一实例 Sim2 模拟器云版（支持稀疏模式）
	CloudBaiduSim2Heaven	使用 C++ 编写的单一实例集群 Sim2 模拟器云版
	CloudBaiduSim2Lake	使用 C++ 编写的单一实例 Sim2 模拟器云版 (支持 GPU 运算)
	CloudAerAtBD	开源 Aer(C++ 版) 模拟器云版
	CloudIoPCAS	来自中科院物理所的 10 比特超导量子计算机 (VIP)
	CloudBaiduQPUQian	来自百度量子计算研究所的 8 比特超导量子计算机 (VIP)
	CloudIonAPM	来自中科院精密测量院的 1 比特离子阱量子计算机 (VIP)

常见问题

1. 如何在量易伏上编程？

   - 用户可以混合使用 Python 编程语言和 QASM 进行实验和开发；
   - 如用户使用 Quantum-hub 上的 PyOnline / QComposer / YunIDE 进行实验，平台自动调用 Token，不需要用户自行填入；
   - 如用户使用 QCompute 进行实验，需要传入 Token 以连接远端模拟器或设备。

2. 为什么要用 QCompute ？

   - 可以迁移使用经典语言知识；
   - 非联网的情况下依然可以使用本地模拟器。

3. 本地模拟器和在线模拟器有什么区别？

   - 本地模拟器使用本地资源，在线模拟器使用百度云计算资源；
   - 在线模拟器在百度云计算的支持下提供更为强大的算力。

4. 如何加速计算过程？

   - 使用在线模拟器；
   - 如果是交互式计算，考虑使用并行计算方法。

5. Credit 的用处？

   - 每次使用云端模拟器运行量子电路消费 1 点；
   - 使用 YunIDE 根据资源和用时也将消费一定点数，消耗速率详见 1.2.5；
   - 每次使用真机运行量子电路消费 1 点；
   - 获取更多点数请通过 Quantum-hub 中的 **【意见反馈】**联系我们。

6. 如何注册 VIP？

   VIP 注册为邀请制，用户需要完成相应身份核验:

   - 提交 VIP 申请到 Quantum-hub 中的 **【意见反馈】**;
   - 待后台判断通过，则申请成功，【个人中心】页显示用户 VIP 级别。

7. Quantum-hub 与 QCompute 的区别和联系？

   - Quantum-hub 是量易伏中控端，无论云端 PyOnline, QComposer, YunIDE 或是 QCompute 用云端资源计算的结果都可以从 Quantum-hub 获取；
   - QCompute 是量易伏的一个高级开发组件，通过此组件，用户可以进行 Python 与简单的量子编程语言的混合编程等。

参考文献

[1] Charles H. Bennett and Stephen J. Wiesner. Communication via one- and two-particle operators on einstein-podolsky-rosen states. *Phys. Rev. Lett.*, 69:2881–2884, Nov 1992.

[2] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1):4213, 2014.