外部构建器和启动器

在 Hyperledger Fabric 2.0 之前，构建和启动链码的过程都是 peer 实现的一部分，不能轻易自定义。Peer 节点上安装的所有链码都将使用 Peer 节点中硬编码的语言特定逻辑来“构建”。该构建过程将生成一个 Docker 容器镜像，该镜像将被启动为一个连接到 Peer 节点的客户端来执行链码。

该方法限制了链码只能由少数语言实现，需要 Docker 作为部署环境的一部分并且阻碍了将链码作为一个长期运行的服务进程。

从 Fabric 2.0 开始，外部构建器和启动器通过支持操作者使用可以构建、启动和发现链码的程序扩展 peer 从而解决了这些限制。要使用这个功能，你需要创建你自己的构建包并修改 peer 的 core.yaml 文件使其包含一个新的 `externalBuilder` 配置元素，该配置可以使 Peer 节点知道外部构建器是可用的。后边的章节详细介绍了这个过程。

注意，如果没有已配置的外部构建器声明链码包，peer 将尝试处理链码包，就好像链码包是由标准的 Fabric 打包工具（比如，peer CLI 或者 node SDK）创建的一样。

注意： 这是一个高级特性，可能会需要自定义的 peer 镜像包。例如，下面的示例中使用的 `go` 和 `bash`，这些并没有包含在官方 `fabric-peer` 镜像中。

外部构建模型

Hyperledger Fabric 外部构建器和启动器不严格地基于 Heroku Buildpacks。一个构件包的实现是简单的程序或脚本的集合，这些程序或脚本会将应用程序构件转换为可以运行的东西。构建包模型适用于链码包，并且扩展支持链码执行和发现。

外部构建器和启动器 API

外部构建器和启动器由四段代码或者脚本组成：

- `bin/detect`： 决定该构建包是否用于构建链码包并启动它。
- `bin/build`： 将链码包转换为可执行链码。
- `bin/release` （可选）： 向 peer 提供链码相关元数据。
- `bin/run` （可选）： 运行链码。

`bin/detect`

`bin/detect` 脚本负责决定该构建包是否用于构建链码包并启动它。peer 调用 `detect` 需要两个参数：

```
bin/detect CHAINCODE_SOURCE_DIR CHAINCODE_METADATA_DIR
```

当调用 `detect` 的时候， `CHAINCODE_SOURCE_DIR` 目录下包含链码源码， `CHAINCODE_METADATA_DIR` 目录包含 `metadata.json` 文件，这些文件都来自 peer 上安装的链码包。 `CHAINCODE_SOURCE_DIR` 和

`CHAINCODE_METADATA_DIR` 应该被视为只读输入。如果构建包要被应用到链码源码包，`detect` 必须返回退出代码 `0`；任何其他退出代码都表示构建包不能被应用。

下边是一个简单的 go 链码 `detect` 脚本示例：

```
#!/bin/bash

CHAINCODE_METADATA_DIR="$2"

# use jq to extract the chaincode type from metadata.json and exit with
# success if the chaincode type is golang
if [ "$(jq -r .type "$CHAINCODE_METADATA_DIR/metadata.json" | tr '[:upper:]' '[:lower:]')" = "gol
    exit 0
fi

exit 1
```

bin/build

`bin/build` 脚本负责构建、编译或者转换链码包的内容为可以让 `release` 和 `run` 使用的构件。peer 调用 `build` 需要三个参数：

```
bin/build CHAINCODE_SOURCE_DIR CHAINCODE_METADATA_DIR BUILD_OUTPUT_DIR
```

当调用 `build` 的时候，`CHAINCODE_SOURCE_DIR` 指向链码源码，`CHAINCODE_METADATA_DIR` 指向 `metadata.json` 文件，这些文件都来自 peer 上安装的链码包。`BUILD_OUTPUT_DIR` 是 `build` 必须放置 `release` 和 `run` 需要的构件的目录。构件脚本应该将 `CHAINCODE_SOURCE_DIR` 和 `CHAINCODE_METADATA_DIR` 看作是只读的，但 `BUILD_OUTPUT_DIR` 是可写的。

当 `build` 以退出代码 `0` 完成时，`BUILD_OUTPUT_DIR` 中的内容将被复制到 peer 管理的持久存储中；任何其他退出代码都认为是失败。

以下是一个简单的 go 链码 `build` 脚本示例：

```
#!/bin/bash

CHAINCODE_SOURCE_DIR="$1"
CHAINCODE_METADATA_DIR="$2"
BUILD_OUTPUT_DIR="$3"

# extract package path from metadata.json
GO_PACKAGE_PATH="$(jq -r .path "$CHAINCODE_METADATA_DIR/metadata.json")"
if [ -f "$CHAINCODE_SOURCE_DIR/src/go.mod" ]; then
    cd "$CHAINCODE_SOURCE_DIR/src"
    go build -v -mod=readonly -o "$BUILD_OUTPUT_DIR/chaincode" "$GO_PACKAGE_PATH"
else
    GO111MODULE=off go build -v  -o "$BUILD_OUTPUT_DIR/chaincode" "$GO_PACKAGE_PATH"
fi

# save statedb index metadata to provide at release
if [ -d "$CHAINCODE_SOURCE_DIR/META-INF" ]; then
    cp -a "$CHAINCODE_SOURCE_DIR/META-INF" "$BUILD_OUTPUT_DIR/"
fi
```

bin/release

`bin/release` 脚本负责向 peer 节点提供链码元数据。`bin/release` 是可选的。如果没有提供，则跳过该步骤。peer 节点使用两个参数调用 `release`：

```
bin/release BUILD_OUTPUT_DIR RELEASE_OUTPUT_DIR
```

当调用 `release` 时，`BUILD_OUTPUT_DIR` 包含由 `build` 程序填充的构建，应该被视为只读输入。`RELEASE_OUTPUT_DIR` 是 `release` 必须放置的由 peer 使用的构建的目录。

`release` 完成后，peer 节点将从 `RELEASE_OUTPUT_DIR` 中消耗两种类型的元数据：

- CouchDB 的状态数据库索引定义
- 外部链码服务器连接信息（`chaincode/server/connection.json`）

如果链码需要 CouchDB 索引定义，`release` 负责将索引放置到 `RELEASE_OUTPUT_DIR` 下的 `statedb/couchdb/indexes` 目录中。索引必须有 `.json` 扩展。 有关详细信息，请参阅 CouchDB indexes 文档。

在使用链码服务器实现的情况下，`release` 负责用与链码服务器地址以及和链码通信所需的任何 TLS 资产来填充 `chaincode/server/connection.json`。当服务器连接信息提供给 peer 节点时，`run` 将不会被调用。 有关详细信息，请参阅 ChaincodeServer 文档。

以下是一个简单的 go 链码 `release` 脚本示例：

```
#!/bin/bash

BUILD_OUTPUT_DIR="$1"
RELEASE_OUTPUT_DIR="$2"

# copy indexes from META-INF/* to the output directory
if [ -d "$BUILD_OUTPUT_DIR/META-INF" ] ; then
   cp -a "$BUILD_OUTPUT_DIR/META-INF/"* "$RELEASE_OUTPUT_DIR/"
fi
```

`bin/run`

`bin/run` 脚本负责运行链码。 peer 节点使用两个参数调用 `run`：

```
bin/run BUILD_OUTPUT_DIR RUN_METADATA_DIR
```

当 `run` 被调用时，`BUILD_OUTPUT_DIR` 包含由 `build` 程序填充的构建，以及一个名为 `chaincode.json` 的文件填充 `RUN_METADATA_DIR`，该文件包含了 chaincode 连接和注册 peer 节点所必需的信息。请注意，`bin/run` 脚本应该将这些 `BUILD_OUTPUT_DIR` 和 `RUN_METADATA_DIR` 目录视为只读输入。`chaincode.json` 中包含的密钥是：

- `chaincode_id`：和链码包关联的唯一 ID。

- `peer_address`：被 peer 节点托管的 `ChaincodeSupport` 的 gRPC 服务器端点的以 `host:port` 为格式的地址。
- `client_cert`：当链码建立与 peer 节点的连接时，必须使用 peer 节点生成的 PEM 编码的 TLS 客户端证书。
- `client_key`：当链码建立与 peer 节点的连接时，必须使用 peer 节点生成的 PEM 编码的客户端密钥。
- `root_cert`：由 peer 服务器托管的 `ChaincodeSupport` 的 gRPC 服务器端点的 PEM 编码的 TLS 根证书。
- `mspid`：peer 节点的本地 mspid。

当 `run` 终止时，peer 节点认为链码终止。如果对链码的另一个请求到达，peer 节点将通过再次调用 `run` 来尝试启动链码的另一个实例。在调用之间不得缓存 `chaincode.json` 的内容。

以下是一个简单的 go 链码 `run` 脚本示例：

```
#!/bin/bash

BUILD_OUTPUT_DIR="$1"
RUN_METADATA_DIR="$2"

# setup the environment expected by the go chaincode shim
export CORE_CHAINCODE_ID_NAME="$(jq -r .chaincode_id "$RUN_METADATA_DIR/chaincode.json")"
export CORE_PEER_TLS_ENABLED="true"
export CORE_TLS_CLIENT_CERT_FILE="$RUN_METADATA_DIR/client.crt"
export CORE_TLS_CLIENT_KEY_FILE="$RUN_METADATA_DIR/client.key"
export CORE_PEER_TLS_ROOTCERT_FILE="$RUN_METADATA_DIR/root.crt"
export CORE_PEER_LOCALMSPID="$(jq -r .mspid "$RUN_METADATA_DIR/chaincode.json")"

# populate the key and certificate material used by the go chaincode shim
jq -r .client_cert "$RUN_METADATA_DIR/chaincode.json" > "$CORE_TLS_CLIENT_CERT_FILE"
jq -r .client_key  "$RUN_METADATA_DIR/chaincode.json" > "$CORE_TLS_CLIENT_KEY_FILE"
jq -r .root_cert   "$RUN_METADATA_DIR/chaincode.json" > "$CORE_PEER_TLS_ROOTCERT_FILE"
if [ -z "$(jq -r .client_cert "$RUN_METADATA_DIR/chaincode.json")" ]; then
    export CORE_PEER_TLS_ENABLED="false"
fi

# exec the chaincode to replace the script with the chaincode process
exec "$BUILD_OUTPUT_DIR/chaincode" -peer.address="$(jq -r .peer_address "$ARTIFACTS/chaincode.jso
```

配置外部构建器和启动器

配置 peer 节点以使用外部构建器涉及在定义外部构建器的 `core.yaml` 中的链码配置块下添加一个外部构建器元素。 每个外部构建器定义必须包括一个名称（用于日志记录）和包含构建器脚本的 `bin` 目录的父目录的路径。

还可以提供一个可选的环境变量名列表，以便在调用外部构建器脚本时从 peer 节点传播。

以下为定义了两个外部构建器的示例：

```
chaincode:
  externalBuilders:
  - name: my-golang-builder
    path: /builders/golang
    propagateEnvironment:
    - GOPROXY
    - GONOPROXY
    - GOSUMDB
```

```
    - GONOSUMDB
  - name: noop-builder
    path: /builders/binary
```

在此示例中，“my-golang-builder“的实现包含在 `/builders/golang` 目录中，其构建脚本位于 `/builders/golang/bin` 中。 当 peer 节点调用与“my-golang-builder”关联的任何构建脚本时，它将只传播在 `propagateEnvironment` 中环境变量的值。

注意：以下环境变量总是传播到外部构建器：

- LD_LIBRARY_PATH
- LIBPATH
- PATH
- TMPDIR

当存在 `externalBuilder` 配置时，peer 节点将按照提供的顺序遍历构建器列表，调用 `bin/detect`，直到成功完成为止。 如果没有构建器成功完成 `detect`，则 peer 节点将返回使用在 peer 节点中实现的遗留 Docker 构建过程。 这意味着外部构建器是完全可选的。

在上面的示例中，peer 节点将尝试使用“my-golang-builder”，其次是“noop-builder”，最后是 peer 内部构建过程。

链码包

作为 Fabric2.0 引入的新生命周期的一部分，链码包格式从序列化协议缓冲区消息更改为 gzip 压缩 POSIX 磁盘存档。 通过 `peer lifecycle chaincode package` 创建的链码包使用此新格式。

生命周期链码包内容

生命周期链码包包含两个文件。 第一个文件 `code.tar.gz` 是一个 gzip 压缩 POSIX 磁带存档。此文件包括链码的源构件。由 peer CLI 创建的包将链码实现源置于 `src` 目录下，链码元数据（如 CouchDB 索引）置于 `META-INF` 目录下。

第二个文件，`metadata.json` 是一个JSON文档，有三个键：

- `type`： 链码类型（例如，GOLANG、JAVA、NODE）
- `path`： go 链码有关的，GOPATH 或者 GOMOD 到 main 链码包的相关的路径；其他类型合约不需要定义
- `label`：用于生成 包 id 的链码标签，新链码生命周期过程通过此包 id 来识别。

请注意，`type` 和 `path` 字段仅由 docker 平台构建使用。

链码包和外部构建器

当链码包安装到 peer 节点时，`code.tar.gz` 和 `metadata.json` 的内容在调用外部构建器之前不被处理，除了被新生命周期进程用来计算包 id 的 `label` 字段。 这为用户就如何打包外部构建器以及启动器处理的源文件和元数据提供了很大的灵活性。

例如，可以构建一个自定义链码包，该包包含 `code.tar.gz` 中的预编译链码实现以及一个 `metadata.json` 文件。 允许 *binary buildpack* 检测自定义包，验证二进制文件的 hash，并将程序运行为链码。

另一个例子是一个链码包，它只包含状态数据库索引定义和外部启动程序连接到正在运行的链码服务器所需的数据。 在这种情况下，`build` 进程只是从进程中提取元数据，并且 `release` 将它呈现给 peer 节点。

唯一的要求是 `code.tar.gz` 只能包含常规文件和目录条目，并且条目不能包含可能导致文件写入链码包逻辑根之外的路径。