以下是对 `artifactory.sh` 脚本关键逻辑部分的详细说明：

一、版本判断逻辑

在软件发布过程中，准确判断版本类型至关重要，它决定了制品的流向和发布策略。

- **正式发布版本判断**：在 `IsDelivery` 函数中，通过比较 `CI_COMMIT_TAG`（代码提交标签）和 `PROJECT_VERSION`（项目版本号）是否相等来判断是否为正式发布版本。
  - **代码逻辑**：`if [[ $CI_COMMIT_TAG == $PROJECT_VERSION ]]; then`，当二者相等时，意味着当前代码提交对应的版本是正式发布版本。此时设置 `info` 变量为“正式发布的版本不上传至 Daily”，这是为了确保正式版本不在日常开发环境（Daily）中上传，保证不同环境的版本隔离。
  - **作用和意义**：这种判断机制可以避免正式版本的制品在开发环境中干扰日常测试和开发工作，同时也能保证正式版本在合适的环境（如生产环境相关的存储库）进行管理和部署。
- **DevRelease 全真版本判断**：`IsIQDelivery` 函数则用于判断是否为发布 DevRelease 的全真版本。它通过检查 `CI_COMMIT_TAG` 是否等于 `PROJECT_VERSION` 加上 `q` 后缀来实现。
- **代码逻辑**：`if [[ $CI_COMMIT_TAG == $PROJECT_VERSION"q" ]]; then`，如果满足此条件，说明是 DevRelease 全真版本，同样设置 `info` 变量为“发布 DevRelease 的全真版本不上传至 Daily”。
- **作用和意义**：这有助于区分特定的内部版本，使其遵循特定的发布规则，不在日常开发环境中上传，保证开发环境的纯净性以及版本管理的准确性。

二、Artifactory 交互逻辑

脚本与 Artifactory 制品库的交互是实现制品上传和管理的核心环节。

- **认证与上传基础**：在各个涉及上传的函数（如 `PublishIntegrate`、`PublishTest` 等）中，都使用了 `curl` 命令，并结合 `ARTIFACTORY_USER` 和 `ARTIFACTORY_TOKEN` 进行身份认证。
  - **代码示例**：以 `PublishIntegrate` 函数为例，在文档上传部分，`curl -u $ARTIFACTORY_USER:$ARTIFACTORY_TOKEN -T $source_path/$doc ${PATH_INTEGRATE}docs/`；在非文档上传部分，`curl -H X-Checksum-sha1:${sha_tar} -u`

`$ARTIFACTORY_USER:$ARTIFACTORY_TOKEN -T ${dist_tar} $PATH_INTEGRATE` 。这里 `-u` 选项用于提供用户名和密码进行认证， `-T` 选项指定要上传的文件。

- **作用和意义**：通过这种认证方式，确保只有授权的用户（由 `ARTIFACTORY_USER` 和 `ARTIFACTORY_TOKEN` 确定）能够将制品上传到 Artifactory 库中，保证了制品库的安全性和访问控制。

- **文档上传逻辑**：当需要上传文档时（以 `PublishIntegrate` 函数中 `type` 为 `publish-integrate-docs` 为例）：
- **代码逻辑**：首先指定源目录 `source_path="formatted_spec"` ，然后通过 `for doc in $(ls $source_path); do` 循环遍历该目录下的所有文件。对于每个文件，使用 `curl -u $ARTIFACTORY_USER:$ARTIFACTORY_TOKEN -T $source_path/$doc ${PATH_INTEGRATE}docs/` 命令将文件上传至 `PATH_INTEGRATE` 路径下的 `docs` 子目录。
- **作用和意义**：这种方式可以将项目相关的文档按照特定的目录结构上传到 Artifactory，方便后续的查阅和管理，同时也保证了文档在制品库中的存储位置规范统一。
- **非文档（制品包）上传逻辑**：对于非文档的制品包上传（如 `PublishIntegrate` 函数中的非文档上传分支）：
- **代码逻辑**：先计算文件的校验和，如 `sha_tar=` 相关计算命令（不同函数中获取方式类似），然后使用 `curl -H X-Checksum-sha1:${sha_tar} -u $ARTIFACTORY_USER:$ARTIFACTORY_TOKEN -T ${dist_tar} $PATH_INTEGRATE` 命令上传制品包。这里 `-H` 选项用于设置请求头，传递文件的校验和信息（ `X-Checksum-sha1` ），确保上传的文件完整性可验证。
- **作用和意义**：通过传递校验和，Artifactory 可以在接收制品包时验证其完整性，防止因网络传输等问题导致文件损坏而未被察觉。同时，将制品包上传到指定路径，便于在后续的发布和部署流程中进行引用和使用。

三、GitLab 集成逻辑

与 GitLab 集成可以在代码仓库中创建发布记录，方便团队跟踪发布状态和制品关联情况。

- **制品准备与上传**：在 `PublishDelivery` 和 `PublishIQDelivery` 函数中，首先进行制品的准备工作。

    - **代码逻辑**：以 `PublishDelivery` 函数为例，先打印目标地址 `echo "Daily的目标地址:$PATH_BETA"` ，然后创建临时目录 `mkdir -p atifactory` ，使用 `wget -r -np -nd --user=$ARTIFACTORY_USER --password=$ARTIFACTORY_TOKEN -P ./atifactory $PATH_BETA` 从指定路径拉取制品到临时目录。接着清理无关文件 `rm -rf ./atifactory/index.html*` ，并定义正式发布版本上传的目标路径 `echo "正式发布的版本上传至 DevRelease:$PATH_DELIVERY_SOURCE"` 。之后通过 `tar_list= ls ./atifactory` 、`` `cd atifactory` `` 等命令遍历临时目录下的文件，计算每个文件的校验和（`` `sha_tar=`shalsum ${tar} | awk '{print $1}' ` ） ，并根据文件格式（是否为 `.tar.gz` ）决定上传路径，使用 `curl` 命令上传至 Artifactory 指定路径（ `PATH_DELIVERY_SOURCE` 相关路径 ）。
    - **作用和意义**：这些步骤确保了要发布的制品从源路径准确拉取到本地，并进行必要的清理和校验，然后上传到 Artifactory 的合适位置，为后续在 GitLab 中创建发布记录提供准确的制品关联信息。
- **创建 GitLab 发布记录**：在制品上传完成后，构建包含制品链接等信息的 JSON 数据，并调用 GitLab API 创建发布记录。
- **代码逻辑**：构建 JSON 数据部分，例如 `data='{ "name": "'"${PROJECT_NAME}"'-'"${PROJECT_VERSION}"'", "tag_name": "'"${PROJECT_VERSION}"'", "description": "'"${PROJECT_NAME}"' release '"${PROJECT_VERSION}"'", "assets": { "links": [ '"${link_list}"' ] } }'` ，这里定义了发布记录的名称、标签名、描述以及关联的制品链接（ `link_list` 中存储了制品的相关链接信息）。然后使用 `curl --header 'Content-Type: application/json' --header "PRIVATE-TOKEN:$GITLAB_TOKEN_READ" --data "$data" --request POST "${GITLAB_ROOT}api/v4/projects/${CI_PROJECT_ID}/releases"` 命令调用 GitLab API 创建发布记录。其中 `--header` 选项用于设置请求头，指定数据格式为 JSON 以及提供 GitLab 访问令牌（ `GITLAB_TOKEN_READ` ） ， `--data` 选项传递构建好的 JSON 数据， `--request POST` 表示使用 POST 请求创建发布记录。
- **作用和意义**：通过在 GitLab 中创建发布记录，将发布的版本信息、制品链接等关键内容进行记录和展示，方便团队成员查看发布历史、关联的制品情况等，有助于更好地管理和跟踪软件发布流程。

四、基于版本类型的发布操作执行

在确定版本类型后，脚本通过主函数 `Main` 以及多个发布函数来执行具体的发布操作：

- **`Main` 函数调度**：`Main` 函数首先调用 `ParseOption` 函数解析传入的命令行参数，获取 `type` 变量的值。然后根据 `type` 的不同值进行条件判断，决定执行何种发布流程：
  - 若 `type` 为 `publish-delivery` ，表示正式交付发布，调用 `PublishDelivery` 函数。此函数会先拉取相关制品到本地临时目录，清理无关文件后，遍历文件计算校验和并上传至 Artifactory 的指定路径（ `PATH_DELIVERY_SOURCE` ），最后构建包含制品链接等信息的 JSON 数据，调用 GitLab API 在 GitLab 中创建发布记录。
  - 若 `type` 为 `publish-iq-delivery` ，表示 DevRelease 全真版本交付发布，调用 `PublishIQDelivery` 函数。流程与 `PublishDelivery` 类似，但在构建 GitLab 发布记录时部分信息（如 `tag_name` 取值依据等）有所不同。
  - 若 `type` 与文档相关，如 `publish-integrate-docs` 、 `publish-test-docs` 等：
    - 当 `type` 为 `publish-integrate-docs` 时，调用 `PublishIntegrate` 函数（文档上传分支）。该函数先判断 `VERSION_MASTER` 是否等于 `PROJECT_VERSION` ，若相等，指定源目录 `source_path="formatted_spec"` ，然后遍历该目录下的文件，使用 `curl` 命令结合 Artifactory 认证信息，将文件上传至 `PATH_INTEGRATE` 路径下的 `docs` 子目录。
    - 当 `type` 为 `publish-test-docs` 时，调用 `PublishTest` 函数（文档上传分支）。此函数先根据 `CI_COMMIT_TAG` 是否符合特定版本格式（ `version_format` 正则匹配 ）计算 `master_version` 和 `test_version` ，确定目标路径 `target_path` ，然后指定源目录 `source_path="formatted_spec"` ，遍历文件并使用 `curl` 上传至目标路径的 `docs` 子目录。
  - 若 `type` 不属于上述明确的交付发布或文档发布类型，则进入另一分支，根据具体 `type` 值（如 `isdelivery` 、 `isiqdelivery` 等）调用相应函数（如 `IsDelivery` 、 `IsIQDelivery` 等）进行判断和处理，或调用不同的发布函数（如 `PublishIntegrate` 、 `PublishTest` 等非文档相关情况）。
- **其他发布函数的条件执行**：除了 `Main` 函数直接调用的发布函数外，在其他发布函数内部也存在基于版本类型判断的逻辑分支：

- 例如在 `PublishIntegrate` 函数中，除了文档上传分支外，还有非文档上传分支。当 `type` 不是 `publish-integrate-docs` 时，会执行非文档上传逻辑。先判断 `VERSION_MASTER` 是否等于 `PROJECT_VERSION` ，若满足条件，计算文件校验和 `sha_tar` ，然后使用 `curl` 命令传递校验和并将制品包（ `dist_tar` ）上传至 `PATH_INTEGRATE` 路径。
- 又如 `PublishTest` 函数中，对于非文档上传的情况，也是先根据 `CI_COMMIT_TAG` 计算版本相关变量，确定目标路径 `target_path` ，然后计算文件校验和，使用 `curl` 命令将制品包上传至目标路径。

五、版本类型与发布操作的关联意义

通过这种根据版本类型执行相应发布操作的机制，能够实现以下目标：

- **环境隔离与版本管理**：确保正式发布版本和特定的全真版本不在日常开发环境（Daily）中上传，实现不同环境的版本隔离，便于版本管理和维护。例如正式版本可以在生产相关环境进行管理和部署，避免在开发环境中造成干扰。
- **发布流程定制化**：针对不同的发布场景（如集成测试、测试、交付等）和制品类型（文档、代码包等），执行特定的上传和发布记录创建操作。这使得发布流程能够根据项目需求进行定制，保证每个版本类型都按照预期的方式进行处理，提高发布的准确性和规范性。
- **提高团队协作效率**：在 GitLab 中创建发布记录，将版本信息和制品关联情况进行记录和展示，方便团队成员查看和跟踪发布历史，有助于提高团队协作效率，确保团队成员对软件发布状态有清晰的了解。